APPENDIX: Cache-Adaptive Analysis (Full Version)

Michael A. Bender [†] Erik D. Demaine[‡] Roozbeh Ebrahimi[§] Jeremy T. Fineman[¶]
Rob Johnson[†] Andrea Lincoln[‖] Jayson Lynch[‡] Samuel McCauley[†]

ABSTRACT

Memory efficiency and locality have substantial impact on the performance of programs, particularly when operating on large data sets. Thus, memory- or I/O-efficient algorithms have received significant attention both in theory and practice. The widespread deployment of multicore machines, however, brings new challenges. Specifically, since the memory (RAM) is shared across multiple processes, the effective memory-size allocated to each process fluctuates over time.

This paper presents techniques for designing and analyzing algorithms in a cache-adaptive setting, where the RAM available to the algorithm changes over time. These techniques make analyzing algorithms in the cache-adaptive model almost as easy as in the external memory, or DAM model. Our techniques enable us to analyze a wide variety of algorithms — Master-Method-style algorithms, Akra-Bazzi-style algorithms, collections of mutually recursive algorithms, and algorithms, such as FFT, that break problems of size N into subproblems of size $\Theta(N^c)$.

We demonstrate the effectiveness of these techniques by deriving several results:

- We give a simple recipe for determining whether common divide-and-conquer cache-oblivious algorithms are optimally cache adaptive.
- We show how to bound an algorithm's non-optimality. We give a tight analysis showing that a class of cache-oblivious algorithms is a logarithmic factor worse than optimal.
- We show the generality of our techniques by analyzing the cache-oblivious FFT algorithm, which is not covered by the above theorems. Nonetheless, the same general techniques can show that it is at most $O(\log \log N)$ away from optimal in the cache adaptive setting, and that this bound is tight.

These general theorems give concrete results about several algorithms that could not be analyzed using earlier techniques. For example, our results apply to Fast Fourier Transform, matrix multiplication, Jacobi Multipass Filter, and cache-oblivious dynamic-programming algorithms, such as Longest Common Subsequence and Edit Distance.

Our results also give algorithm designers clear guidelines for creating optimally cache-adaptive algorithms.

This research was supported in part by NSF grants CCF 1114809, CCF 1217708, CCF 1218188, CCF 1314633, IIS 1247726, IIS 1251137, CNS 1408695, and CCF 1439084; MADALGO - Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation; and a Stanford Graduate Fellowship.

[†]Department of Computer Science, Stony Brook University, Stony Brook, NY 11794-4400, USA. Email: {bender, rob, smccauley}@cs.stonybrook.edu.

[‡]MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar Street, Cambridge, MA 02139, USA. Email: {edemaine,jaysonl}@mit.edu.

[§]Google Inc., 1600 Amphitheatre Parkway, Mountain View, CA 94043 USA. Email: rebrahimi@google.com.

[¶]Department of Computer Science, Georgetown University, 37th and O Streets, N.W., Washington D.C. 20057, USA. Email: jfineman@cs.georgetown.edu.

[‖]Department of Computer Science, Stanford University, Palo Alto, CA 94305 USA. Email: andreali@cs.stanford.edu.

SPAA '16, July 11 - 13, 2016, Pacific Grove, CA, USA

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4210-0/16/07...$15.00

DOI: http://dx.doi.org/10.1145/2935764.2935798

1. INTRODUCTION

Memory fluctuations are the norm on most computer systems. Each process's share of memory changes dynamically as other processes start, stop, or change their own demands for memory. This phenomenon is particularly prevalent on multi-core computers.

External-memory computations especially suffer from these fluctuations. Examples include:

- joins and sorts in a database management system (DBMS),
- irregular, I/O-bound, shared-memory parallel programs,
- cloud computing services running on shared hardware,

and essentially any external-memory computation running on a time-sharing system.

Database and scientific computing researchers and practitioners have recognized this problem for over two decades [10, 22, 23], and have developed many sorting and join algorithms [19, 24, 25, 31–33] that offer good empirical performance when memory changes size dynamically. However, most of these algorithms are designed to perform well in the common case, but perform poorly in the worst case [4, 5].

In contrast to this reality, most of today's performance models for external-memory computation assume a *fixed internal memory size* M (see, e.g., [29]) and hence algorithms designed in these models cannot cope when M changes. This means that most external-memory algorithms cannot take advantage of memory freed by other processes, and can begin thrashing if the system takes back too much memory.

Thus, there is a gap between the state of the world, where memory fluctuations are the rule, and today's tools for designing and analyzing external-memory algorithms, which assume fixed internal-memory sizes.

Barve and Vitter [4, 5] took the first major step towards closing this gap by showing that worst-case, external-memory bounds are possible in an environment where the cache[1] changes size. Barve and Vitter generalized the DAM (disk-access machine) model [1] to allow the memory size M to change periodically. They give optimal algorithms for sorting, FFT, matrix multiplication, LU decomposition, permutation, and buffer trees. Their work shows that it is possible to specially design intricate algorithms to handle adaptivity, but stops short of giving a general framework.

Bender et al. [8] took the next major step towards closing this gap between theoretical performance models and real systems. They formally define the cache-adaptive model,[2] and prove that some—but not all—optimal cache-oblivious[3] algorithms [17, 18, 27] remain optimal when the cache changes size dynamically. Specifically, if a recursive cache-oblivious algorithm performs $O(1)$ block transfers in addition to its recursive calls, then it is optimally cache adaptive. So is lazy funnel sort [9], despite not fitting this recursive pattern. Bender et al.'s results are encouraging. Because cache-oblivious algorithms are well understood, frequently easy to design, and widely deployed, there is hope that provably good cache-adaptive algorithms can also be deployed in the field.

On the other hand, open questions remain. In particular, the primary contribution of Bender et al. [8] was proposing a computational model rather than an analysis technique. Is there an algorithmic toolkit for cache-adaptive analysis so that future engineers could write their own cache-oblivious algorithms and then quantify their adaptivity? How can we analyze more general forms of recursive algorithms, e.g., in the common case where the additive term is $\omega(1)$? (Examples include cache-oblivious FFT, some versions of cache-oblivious matrix multiply and mutually recursive cache-oblivious dynamic programming algorithms such as LCS [11], Edit Distance [11], and Jacobi Multipass Filter [27].) How can we prove that a recursive algorithm is *not* optimally cache adaptive? For such algorithms, how far are they from optimality? The point of the present paper is to help answer these and other questions.

[1]We use "cache" to refer to the smaller level in any two-level hierarchy. Because this paper emphasizes RAM and disk, we use the terms "internal memory," "RAM," and "cache" interchangeably.

[2]The cache-adaptive model allows memory to change size more rapidly and unpredictably than the model of [4, 5], meaning that cache-adaptive results are more likely to hold up in the real world.

[3]***Cache-oblivious*** algorithms are not parameterized by the memory hierarchy, yet they often achieve provably optimal performance for any *static* hierarchy; see Section 2.

Results

The contribution of this paper is a set of tools that make analyzing the performance of recursive algorithms in the cache-adaptive setting almost as easy as analyzing their performance in the DAM [1] or cache-oblivious/ideal-cache models [17, 18, 27]. Analyzing the performance of many algorithms in the cache-adaptive model boils down to mechanically transforming the recurrence relation for the algorithm's I/O complexity, solving this new recurrence, and comparing the result to a problem-specific lower bound: if these bounds are asymptotically equal, then the algorithm is optimal; if they are not, then their ratio bounds how far the algorithm is from optimal.

Our techniques are general. They can analyze a wide variety of algorithms: Master-method-style algorithms [13], Akra-Bazzi-style algorithms [2], algorithms made of several mutually recursive functions, and algorithms, such as cache-oblivious FFT, that break a problem of size N into subproblems of size $\Theta(N^c)$.

Our results provide easy guidelines for cache-adaptive-algorithm designers. For example, in the case of linear-space-complexity Master-method-style algorithms, our results give an easy rule: if you want your algorithm to be optimal in the cache-adaptive model, then it can perform linear scans of size up to $O(N^c)$, where $c < 1$, in addition to its recursive calls. See Theorem 7.5 for the detailed criteria.

Our results suggest that the problem of designing and analyzing algorithms that adapt to memory fluctuations is tractable. The cache-adaptive model places almost no restrictions on how much and when memory can change size, so results in the model can carry over to most real-world systems. Despite this generality, our method enables algorithm designers to easily evaluate performance in this model.

We demonstrate our techniques by deriving several concrete results. (Below N refers to the input size.)

- We establish that cache-obliviousness does not always lead to cache-adaptivity using a variation of the cache-oblivious naïve matrix multiplication algorithm of Frigo et al. [17]. While this variation is optimal in the DAM model, it is a $\Theta(\log N)$ factor away from optimal in the cache-adaptive model. This result serves as a concrete example of our more general techniques.
- We completely characterize when a Master-method-style linear-space-complexity algorithm is optimal in the cache-adaptive model; see Theorem 7.5. We show that when such an algorithm is DAM-optimal, it is at most an $O(\log N)$-factor slower than optimal in the cache-adaptive model.
- More generally, we completely characterize when a set of mutually recursive linear-space-complexity Akra-Bazzi-style algorithms are optimal in the cache-adaptive model (Theorem 6.12 and Theorem 6.13).
  We apply these theorems to show the cache-adaptive optimality of mutually recursive algorithms such as the cache-oblivious dynamic programming algorithms of Chowdhury and Ramachandran [11] for Longest Common Subsequence and Edit Distance, and Prokop's cache-oblivious Jacobi Multipass Filter Algorithm [27].
- We show that the same techniques can be used to analyze non-Akra-Bazzi-style algorithms. As an example, we show that the cache-oblivious FFT algorithm [17] is $O(\log \log N)$ away from optimal.

Paper overview.

The rest of the paper is organized as follows. Section 2 reviews the cache-adaptive model, including the square memory profile and tools for analyzing cache-adaptive algorithms.

In Section 3, we present an axiomatization of an algorithm's *progress* in solving a problem in the cache-adaptive model. This axiomatization is simple, intuitive and easy to work with. We state almost all of our optimality criteria theorems (in Section 7) in terms of problems that have *progress* functions associated with them and these progress functions should satisfy our axiomatization in Section 3.

Section 3 lays the groundwork for proving that an algorithm is optimal or within some factor of optimal, by capturing both the computational needs of a problem and the maximum progress rate of an algorithm.

In Section 4, we establish that cache-obliviousness does not always lead to cache-adaptivity. We prove that a variation of the cache-oblivious naïve matrix multiplication algorithm of Frigo et al. [17], MM-Scan, that is optimal in the DAM model, is a $\Theta(\log N)$ factor away from being optimal in the cache-adaptive model when solving problem instances of size N.

In Section 7 we characterize optimality criteria for several classes of recursive cache-oblivious algorithms. These classes include Master Method style recursion [13], Akra-Bazi style recursions [3] and classes of multiple/mutual recursions composed from Akra-Bazi style recursions.

Later in Section 8, we provide tools to explicitly *derive* progress functions for problems. To this end, we use machinery from lower bound proof techniques of the DAM model, like the *red-blue pebble game* of Hong and Kung [20], the *red pebble game* of Savage [28], and the techniques of Aggarwal and Vitter [1]. We hope that the tools presented in Section 8 allow for a seamless *porting* of optimality analysis in the DAM model to the optimality analysis in the cache-adaptive model.

Section 9 applies the theorems proved in Section 7 together with explicit progress bounds derived in Section 8 to exhibit the cache-adaptive optimality of several cache-oblivious algorithms like the dynamic dynamic programming algorithms of Chowdhury and Ramachandran [11] for Longest Common Subsequence (LCS) and Edit Distance problems, and the cache-oblivious Jacobi Multipass Filter algorithm of Prokop [27].

In Section 10, we show that the analytic techniques of Section 7 can be used to analyze algorithms that don't fit the Akra-Bazzi form. As an example, we show that the cache-oblivious FFT algorithm of Frigo et al. [17] is a $O(\log \log N)$ factor away from being optimal when solving problem instances of size N.

2. CACHE-ADAPTIVE MODEL, DEFINITIONS, AND ANALYTICAL TOOLS

The ***cache-adaptive*** model [8] is the same as the DAM model [1] except that memory may change size after each I/O (i.e., after each cache miss).[4] Thus, the size of memory is not a constant, but rather a function $m(t)$ giving the size of memory (in blocks) after the tth I/O. We also use $M(t) = B \cdot m(t)$ to represent the size, in words, of memory at time t. We call $m(t)$ and $M(t)$ ***memory profiles*** in blocks and words, respectively.

Optimality in the cache-adaptive model mirrors optimality in the DAM model. On every memory profile, an optimal algorithm has worst-case I/O complexity within a constant factor of any other algorithm's worst-case I/O complexity.

However, in the cache-adaptive model it does not make sense to compare two algorithms' running times directly. That is, given a profile $m(t)$ and two algorithms A and B, we cannot compare A and B's performance by simply running them on $m(t)$ and comparing their running times. To see why, suppose B performs one dummy I/O and then runs A. By any reasonable definition, B's running time should be asymptotically no worse than A's. But consider an input I and profile $m(t)$ that drops to very little memory as soon as $A(I)$ finishes. Now B may finish arbitrarily later than A.

Thus, we formalize optimality by comparing algorithms using ***speed augmentation***. Rather than granting an algorithm extra time, we allow it to perform multiple I/Os per time step in our analysis. Speed augmentation is a theoretical tool to ensure that algorithms are compared on profiles that provide the same overall resources up to a constant factor. Thus, rather than saying one algorithm is (say) twice the speed of another, we say that it performs equally well on hardware with half the latency. This gives the same intuition as classic asymptotic analysis while being meaningful in the cache-adaptive model.

Definition 2.1. *Giving an algorithm A,* ***c-speed augmentation*** *means that A may perform c I/Os in each step of the memory profile.*

In order to make the definition of optimality in the CA model as strong as possible, we allow algorithms to query $m(t)$, even into the future. Thus, an optimal algorithm in the CA model is asymptotically as fast as any other algorithm, even one that can see the future size of memory and plan accordingly.[5]

However, allowing algorithms to query the memory profile creates a problem: a P/poly algorithm may be able to use the memory profile as a "reference string" to speed up its computations on some memory profiles. We rule out this behavior by only permitting ***memory-monotone*** algorithms:

Definition 2.2. *A* ***memory monotone*** *algorithm runs at most a constant factor slower when given more memory.*

Memory monotonicity is a weak restriction: all cache-oblivious algorithms are memory monotone, as are LRU, the optimal offline paging algorithm, and many other paging algorithms. Memory monotone also includes almost all "reasonable" DAM-model algorithms. One notable exception is FIFO paging [7], which was recently shown not to be memory monotone [16].

Definition 2.3. *An algorithm A that solves problem P is* ***optimal*** *in the cache-adaptive model if there exists a constant c such that on all memory profiles and all sufficiently large input sizes N, the worst-case running time of a c-speed-augmented A is no worse than the worst-case running time of any other (non-augmented) memory-monotone algorithm.*

[4]We use I/Os as a proxy for time because we are studying I/O-bound algorithms—the algorithm spends most of its time performing these I/Os (see [8]).

[5]Note that while our model allows the algorithms to see the future memory, the cache-oblivious algorithms presented here do not take advantage of this—in fact, they are not even aware of the present size of memory.

As in the DAM model, memory augmentation is needed to show that LRU is constant competitive. We also use memory augmentation to simplify our analyses.

Definition 2.4. *For any memory profile m, we define a* ***c-memory augmented*** *version of m as the profile $m'(t) = cm(t)$. Running an algorithm A with* ***c-memory augmentation*** *on the profile m means running A on the c-memory augmented profile of m.*

If an algorithm is not parameterized by M or B, then we say it is ***cache-oblivious***. These algorithms are designed to perform well without knowing the size of memory.

When analyzing cache-oblivious algorithms in the CA model, we assume that the system performs automatic page replacement. Belady's algorithm [6] turns out to be an optimal offline paging algorithm in the CA model and the least-recently-used (LRU) policy is $O(1)$-competitive with speed and memory augmentation [8].

The performance bounds for cache-oblivious algorithms commonly rely on a so-called ***tall-cache*** assumption, which means that there is a value $H(B)$, polynomial in B, such that $M \geq H(B)$. For example, for cache-oblivious sorting or matrix transpose, $H(B) = \Theta(B^2)$ [17]. We support these kinds of analyses in the CA model as follows.

Definition 2.5. *In the CA model, we say that a memory profile M is* ***H-tall*** *if for all $t \geq 0$, $M(t) \geq H(B)$.*

Definition 2.6. *Algorithm A has* ***space complexity*** *$f(N)$ if for all problems of size N, the number of distinct memory locations accessed by A while processing I is $\Theta(f(N))$.*

Space complexity is often defined as the maximum memory location indexed. This definition is slightly more general.

3. PROGRESS BOUNDS: HOW MUCH AN ALGORITHM CAN DO ON A PROFILE

This section axiomatizes the notion of ***progress bounds***, which are used in many lower-bound proofs in the DAM model, and develops tools to easily port progress bounds from the DAM model to the CA model.

In the DAM model, a ***progress bound*** $\rho(M, T)$ for a problem P gives an upper bound on the amount of progress that any algorithm can make towards solving an instance of P given M words of memory and T I/Os. A ***progress requirement function*** $R(N)$ gives a lower bound on the amount of progress any algorithm must make in order to solve all problem instances of size N. In the DAM model, the I/O complexity of any algorithm must be at least $\Omega(T \cdot R(N)/\rho(M, T))$.

Example 3.1. *The DAM sorting lower bound [1] says that, after sorting each block, a comparison-based sorting algorithm learns at most $O(B \log(M/B))$ bits of information per I/O, given M memory, and must learn $\Omega(N \log N)$ bits to sort. Thus, $R(N) = \Omega(N \log N)$ and $\rho(M, T) = O(TB \log(M/B))$. Hence sorting in the DAM model takes at least $R(N)/\rho(M, 1) = \Omega(\frac{N}{B} \log_{M/B} N)$ I/Os.*

Example 3.2. *The DAM matrix multiplication lower bound [20, 21, 28] states that, given M memory and M/B I/Os, no naive matrix multiplication algorithm can perform more than $O(M^{3/2})$ elementary multiplications, and multiplying two $\sqrt{N} \times \sqrt{N}$ matrices requires performing $\Theta(N^{3/2})$ elementary multiplications. Thus $\rho(M, M/B) = M^{3/2}$ and $R(N) = \Theta(N^{3/2})$. Consequently, multiplying two $\sqrt{N} \times \sqrt{N}$ matrices requires $\frac{M}{B} R(N)/\rho(M, \frac{M}{B}) = \Omega(MN^{3/2}/(BM^{3/2})) = \Omega(\frac{N^{3/2}}{B\sqrt{M}})$ I/Os.*

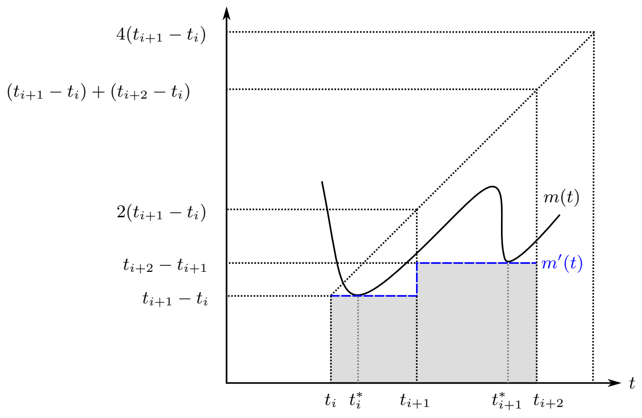

Figure 1: The inner square profile of memory profile $m(t)$. The inner square boundaries comprise t_i, t_{i+1}, and t_{i+2}.

We first generalize the notion of progress bounds to arbitrary profiles. Given an arbitrary memory profile $M(t)$, we write $\rho(M)$ as the upper bound on progress that any algorithm can make on an instance of problem P when given memory profile $M(t)$.

Most[6] progress bounds developed in the DAM model also apply to the CA model because they are "memory-less", i.e., the bound $\rho(M, t)$ applies no matter what the size or content of memory was before the t I/Os in question. In fact, in Section 8, we give a general method for deriving progress bounds from DAM lower bounds based on the red pebble game [28].

The only challenge to using DAM progress bounds on arbitrary memory profiles in the CA model is that some bounds apply only to time-spans during which memory does not change size. For example, the matrix multiply progress bound is defined only for M/B I/Os during which memory is always of size M.

We solve this problem using ***square profiles***. A profile $m(t)$ is square if time can be decomposed into a sequence of regions $t_0 < t_1 < \ldots$ such that $m(t) = t_{i+1} - t_i$ for all $t \in [t_i, t_{i+1})$. Square profiles are useful because we can apply progress bounds, like the matrix multiply progress bound, to each square independently and sum to get a bound on the progress that any algorithm can make on the entire profile. We use the notation $\square_M$ to denote a square of size M words by M/B I/Os, and we denote a square profile with squares of size $M_1, \ldots, M_k$ as $\square_{M_1} \| \cdots \| \square_{M_k}$. We generalize ρ from the DAM model to squares as $\rho(\square_M) = \rho(M, M/B)$.

Definition 3.3. *A memory profile m is* ***square*** *if there exist boundaries $0 = t_0 < t_1 < \ldots$ such that for all $t \in [t_i, t_{i+1})$, $m(t) = t_{i+1} - t_i$. In other words, a square memory profile is a step function where each step is exactly as long as it is tall.*

We can then generalize these bounds to arbitrary profiles by using ***inner square profiles*** [8]. The inner square profile of a profile M is constructed by greedily packing the

[6]In fact, all that we have found.

largest-possible squares under M from left to right, as shown in Figure 1. Square profiles enable us to compute bounds on the amount of progress that any algorithm can make on an arbitrary memory profile.

Definition 3.4. *For a memory profile m, the **inner square boundaries** $t_0 < t_1 < t_2 < \dots$ of m are defined as follows: Let $t_0 = 0$. Recursively define t_{i+1} as the largest integer such that $t_{i+1} - t_i \leq m(t)$ for all $t \in [t_i, t_{i+1})$. The **inner square profile** of m is the profile m' defined by $m'(t) = t_{i+1} - t_i$ for all $t \in [t_i, t_{i+1})$ (see Figure 1).*

Bender et al. [8] proved the following useful lemma about inner square profiles.

Lemma 3.5. *Let m be a memory profile where $m(t+1) \leq m(t) + 1$ for all t. Let $t_0 < t_1 < \dots$ be the inner square boundaries of m, and m' be the inner square profile of m.*

1. *For all t, $m'(t) \leq m(t)$.*
2. *For all i, $t_{i+2} - t_{i+1} \leq 2(t_{i+1} - t_i)$.*
3. *For all i and $t \in [t_{i+1}, t_{i+2})$, $m(t) \leq 4(t_{i+1} - t_i)$.*

The following definitions axiomatize two technical but obvious properties of progress functions: (1) that profiles with more time and memory can support more progress, and (2) that the progress possible on a square profile is just the sum of the progress possible on each of its squares.

Intuitively, we say that one profile is ***smaller*** than another, i.e., offers less memory and/or time, if it can be cut into pieces, each of which fits underneath a corresponding piece of the other.

Definition 3.6. *Let M and U be any two profiles of finite duration. We say that **M is smaller than U, $M \prec U$,** if there exist profiles $L_1, L_2 \dots L_k$ and $U_0, U_1, U_2 \dots U_k$, such that $M = L_1 \| L_2 \dots \| L_k$ and $U = U_0 \| U_1 \| U_2 \dots \| U_k$, and for each $1 \leq i \leq k$,*

(i) *If d_i is the duration of L_i, U_i is a profile with duration $\geq d_i$.*

(ii) *As standalone profiles, L_i is always below U_i.*

Definition 3.7. *A function $\rho : \mathbb{N}^* \to \mathbb{N}$ is **monotonically increasing** if for any profiles M and U, $M \prec U$ implies $\rho(M) \leq \rho(U)$.*

Second, we assume that the progress on a square profile should be, essentially, the sum of the progress possible on each square.

Definition 3.8. *Let $M_1 \| M_2$ indicate concatenation of profiles M_1 and M_2. A monotonically increasing function $\rho : \mathbb{N}^* \to \mathbb{N}$ is **square-additive** if*

(i) *$\rho(\square_M)$ is bounded by a polynomial in M,*

(ii) *$\rho(\square_{M_1} \| \cdots \| \square_{M_k}) = \Theta(\sum_{i=1}^{k} \rho(\square_{M_i}))$.*

Observation 3.9. *If a function ρ is square-additive, then $\rho(\square_N) = \Omega(N)$, because we can fit $\Theta(N/B)$ squares of size B inside a square of size N.*

With these requirements in mind, we can axiomatize the notion of progress bound.

Definition 3.10. *A problem has a **progress bound** if there exists a monotonically increasing polynomial-bounded **progress-requirement function R** : $\mathbb{N} \to \mathbb{N}$ and a square-additive **progress limit function ρ** : $\mathbb{N}^* \to \mathbb{N}$ such that: For any profile M, if $\rho(M) < R(N)$, then no memory-monotone algorithm running under profile M can solve all problem instances of size N.*

We also refer to the progress limit function ρ as the ***progress function*** or ***progress bound***.

Observation 3.11. *If algorithm A has linear space complexity, then $R(N) = O(\rho(\square_N))$.*

3.1 Optimally-Progressing Algorithms

In this paper, we prove that algorithms are optimal (or non-optimal) by analyzing whether they always make within a constant factor of the maximum possible progress on a profile. An algorithm is ***optimally progressing*** if, for every usable profile m that is just long enough for the algorithm to solve all problems of size N, $\rho(m) = O(R(N))$. We first define what it means for a profile to be "usable" and "just long enough" and then define optimally progressing formally.

The CA model allows memory to increase or decrease arbitrarily from one time step to the next. However, since an algorithm can only load one block into memory per time step, its memory usage can only increase by one block per time step.

Definition 3.12. *An h-tall memory profile m is **usable** if $m(0) = h(B)$ and if m increases by at most 1 block per time step, i.e. $m(t+1) \leq m(t) + 1$ for all t.*

Definition 3.13. *For an algorithm A and problem instance I we say a profile M of length ℓ is **I-fitting** if A requires exactly ℓ time steps to process input I on profile M. A profile M is **N-feasible for A** if A, given profile M, can complete its execution on all instances of size N. We further say that M is **N-fitting for A** if it is N-feasible and there exists at least one instance I of size N for which M is I-fitting. (When A is understood, we will simply say that M is N-feasible, N-fitting, etc.)*

Definition 3.14. *For an algorithm A, integer N, and N-feasible profile $M(t)$, let $M_N(t)$ denote the N-fitting prefix of M. We say that algorithm A with tall-cache requirement H is **optimally progressing with respect to a progress bound ρ** (or simply optimally progressing if ρ is understood) if, for every integer N and N-feasible H-tall usable profile M, $\rho(M_N) = O(R(N))$. We say that A is **optimally progressing in the DAM model** if, for every integer N and every constant H-tall profile M, $\rho(M_N) = O(R(N))$.*

The following two lemmas show that usable profiles and square profiles support essentially the same amount of progress. This implies that, if a memory-monotone algorithm is optimally progressing on all usable profiles, then it is optimally progressing on all square profiles, and vice versa. This enables us to focus exclusively on square profiles, which are easier to analyze, when proving algorithms optimal (or non-optimal) in the CA model.

Lemma 3.15. *If ρ is square additive and M is a usable profile with inner square profile M', then $\rho(M) = \Theta(\rho(M'))$.*

Proof. Since ρ is monotonic and $M'(t) \leq M(t)$ for all t, $\rho(M') \leq \rho(M)$. Let $M'_{4,4}$ be the 4-speed and 4-memory augmented version of M'. Since ρ is square-additive and since $\rho(\square_N)$ is bounded by a polynomial in N, $\rho(M'_{4,4}) = O(\rho(M'))$.

We prove that $M \prec M'_{4,4}$. Thus, by monotonicity of ρ,

$$\rho(M') \leq \rho(M) \leq \rho(M'_{4,4}) = O(\rho(M')),$$

which means that $\rho(M) = \Theta(\rho(M'))$.

Let $M[S_i]$ denote the profile M restricted to the interval S_i. Let $k+1$ be the number of inner squares in M'. Define $L_1 = M[S_1 \cup S_2], L_2 = M[S_3], \ldots, L_k = M[S_{k+1}]$, and note that $M = L_1 \| L_2 \| \ldots \| L_k$. Also, define U_i to be a 4-speed 4-memory augmented version of square S_i and allow $U'_k = U_k \| U_{k+1}$. Notice that $M'_{4,4} = U_1 \| U_2 \| \ldots U_{k-1} \| U'_k$.

In order to prove that $M \prec M'_{4,4}$, we show that each U_i, $1 \leq i \leq k-1$ satisfies the three conditions of Definition 3.6, and U'_k satisfies the first two conditions of Definition 3.6.

We start by considering U_1 and L_1. If M is $H(B)$-tall, by Definition 3.12 we have that $M(0) = H(B)$. By Definition 3.4, we have that $t_1 = H(B)$ and since $m(t+1) \leq m(t) + 1$, we have that for all $t \in [0, t_1)$, $M(t) \leq 2t_1$. Moreover, by Lemma 3.5, we know that S_2 is at most twice as long as S_1 and for all $t \in [t_1, t_2)$, $M(t) \leq 4(t_1 - t_0) = 4|S_1|$. Hence, $t_2 \leq 3t_1$, and for all $t \in [0, t_2)$, $M(t) \leq 4|S_1|$. Because U_1 is a 4-speed 4-memory augmented version of S_1, we have that (i) U_1 has a longer duration than $L_1 = M[S_1 \cup M_2]$, and that (ii) L_1 is below U_1.

Similarly, for each U_i, $2 \leq i \leq k$, by Lemma 3.5, we know that S_{i+1} is at most twice as long as S_i and for all $t \in [t_{i+1}, t_{i+2})$, $M(t) \leq 4(t_{i+1} - t_i) = 4|S_i|$. Because U_i is a 4-speed 4-memory augmented version of S_i, we have that (i) U_i has a longer duration than $L_i = M[S_{i+1}]$, and that (ii) L_i is below U_i.

By repeating the above argument for U_k, we see that (i) U_k has a longer duration than $L_k = M[S_{k+1}]$, and that (ii) L_k is below U_k. This means that $U'_k = U_k \| U_{k+1}$ also satisfies both of the above conditions. Therefore, we have shown that $M \prec M'_{4,4}$ and the lemma is complete. $\square$

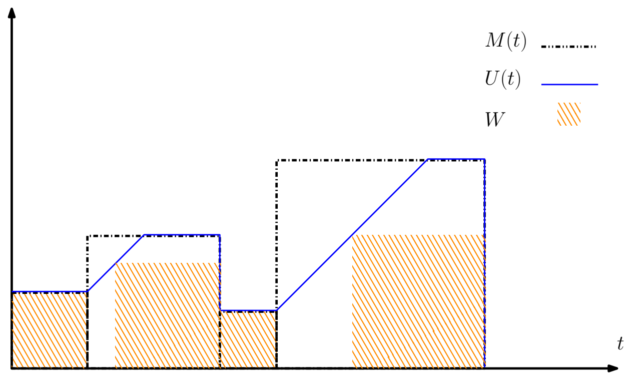

Figure 2: The usable profile beneath each square profile.

Lemma 3.16. *Let ρ be square additive. For every H-tall square memory profile M, there exists a* usable *memory profile U below M such that $\rho(U) = \Theta(\rho(M))$.*

Proof. Let $M = \square_{M_1} \| \square_{M_2} \| \ldots \square_{M_k}$ be any square profile. We construct a usable profile U as follows. We set $U(x) = H(B) + x$ for $x \in [0, \leq M_1 - H(B)]$ and $U(x) = M_1$ for $x \in (M_1 - H(B), M_1)$. For $i \geq 2$, U does the following on the interval $[t_i, t_i + M_i)$ covered by the ith square of M,

(i) If $M_i \leq M_{i-1}$, we let $U(x) = M_i$ for $x \in [t_i, t_i + M_i)$.

(ii) If $M_i > M_{i-1}$, let $U(x) = M_{i-1} + x$ for $x \in [t_i, t_i + M_i - M_{i-1}]$ and $U(x) = M_i$ for $x \in (t_i + M_i - M_{i-1}, t_i + M_i)$.

See Figure 2 for an illustration. Clearly we have $U \prec M$, so by monotonocity of ρ, we have that $\rho(U) \leq \rho(M)$.

We now argue that $\rho(U) = \Omega(\rho(M))$. To exhibit this, we show that there exist mutually disjoint squares $\square_{W_i}$, that all fit below U and for each i, $\square_{W_i}$ is at most 2 times shorter than $\square_{M_i}$. Since each $\square_{W_i}$ fits below U, we have that $W \prec U$ where $W = \square_{W_1} \| \square_{W_2} \| \ldots \square_{W_k}$. On the other hand, since ρ is square-additive, ρ is bounded by a polynomial and thus $\rho(\square_{W_i}) = \Theta(\rho(\square_{M_i}))$. Square-additivity of ρ also means that $\rho(W) = \Theta(\rho(M))$. Since $\rho(W) \leq \rho(U)$ the statement follows.

It remains to show that such $\square_{W_i}$ exist for each i. For each i, if $\square_{U_i} = \square_{M_i}$ (as in case (i) above), we allow $\square_{W_i} = \square_{M_i}$. Otherwise (as in case (ii) above), we let $\square_{W_i}$ be a square that is grown from the rightmost point of $\square_{M_i}$ diagonally to left until it touches U, see Figure 2. Note that because U increases linearly at the beginning of $\square_{M_i}$ until it reaches M_i, the point of $\square_{W_i}$ intersecting U is always on or above the diagonal of $\square_{M_i}$. Therefore, the height of W_i is at least 1/2 the height of M_i. $\square$

Thus, between Lemma 3.15 and Lemma 3.16 we are able to use square profiles for both upper and lower bounds without loss of generality.

Finally, our proofs focus on showing that an algorithm is optimally progressing. This lemma justifies this focus—showing that an algorithm is optimally progressing is sufficient to show that it is cache-adaptive.

Lemma 3.17. *If an algorithm A is optimally progressing, then it is optimally cache adaptive.*

Proof. Let N be a sufficiently large input size. Suppose M is an N-fitting profile for some other algorithm $\mathcal{E}$. With some (unknown and possibly $\Omega(1)$) speed augmentation c, A can solve all problems of size N on M'. Let M'_c be the c-speed augmented version of M', where c is chosen to be as small as possible, so that M'_c is N-fitting for A.

M'_c replaces each square in M' with c squares of the same height, therefore by square-additivity of ρ we have that

$$\rho(M'_c) = c\Theta(\rho(M')).$$

Since A is optimally progressing and M'_c is N-fitting $\rho(M'_c) = O(R(N))$. We have

$$\rho(M'_c) = c\Theta(\rho(M')) = c\Theta(\rho(M)) = O(R(N)).$$

On the other hand, since M is N-fitting for $\mathcal{E}$, we have that $\rho(M) \geq R(N)$, so it must be the case that $c = O(1)$.

We have that with $c = O(1)$ speed augmentation, A can solve all problems of size N in M'. Because M is always above its inner square profile M', and A is memory-monotone (ref. to Definition 2.2), A on M is no more than $f = O(1)$ times slower than A on M'. Therefore, with $cf = O(1)$ augmentation, A can solve all problems of size N on M and hence has a running time no worse than $\mathcal{E}$ on M. $\square$

While optimally progressing is sufficient for adaptivity, we do not know if it is necessary—it remains an open question if there are algorithms that are optimally cache-adaptive but not optimally progressing. On the other hand, *progress optimality* and *competitive optimality* are equivalent in the DAM model.

Lemma 3.18. *If A is an optimal algorithm in the DAM model for a problem P with a progress bound ρ, then A is optimally progressing with respect to ρ in the DAM model.*

Proof. Since, M is an N-fitting profile for A, we must show that $\rho(M) = O(R(N))$.

Let $e_{A,M,N}$ be the minimum time (number of I/Os) required for A to solve all instances of size N given a memory of size M in the DAM model. Consider the finite profile

$$M(t) = \begin{cases} M & \text{if } t \le e_{A,M,N}, \\ 0 & \text{otherwise.} \end{cases}$$

Consider any other algorithm A' for P in the DAM model. Since A is optimal in the DAM model, we have that there exists a constant c, such that for all A', $e_{A,M,N} \le ce_{A',M,N}$.

Among all other algorithms, take algorithm A'' to be the algorithm which minimizes $e_{A',M,N}$. Let

$$M''(t) = \begin{cases} M & \text{if } t \le e_{A'',M,N}, \\ 0 & \text{otherwise.} \end{cases}$$

$$S(t) = \begin{cases} M & \text{if } t \le e_{A'',M,N} - 1, \\ 0 & \text{otherwise.} \end{cases}$$

By Definition 3.10, we have that $\rho(M'') \ge R(N)$. Moreover, since $e_{A'',M,N}$ is minimal among all algorithms, we have that no algorithm can solve all problems of size N on profile $S(t)$. Again, by Definition 3.10, $\rho(S) < R(N)$.

Allow $Q = (S \| S)$ and note that $M'' \prec S$. We have that

$$\rho(M'') \le \rho(Q) = \Theta(\rho(S) + \rho(S)) = O(R(N)).$$

Let $d \ge 1$ be the constant where $e_{A,M,N} = de_{A'',M,N}$, and let

$$M''_d(t) = (\underbrace{M'' \| \ldots M''}_{\lceil d \rceil \text{ times}}) = \begin{cases} M & \text{if } t \le \lceil d \rceil e_{A'',M,N}, \\ 0 & \text{otherwise.} \end{cases}$$

By square additivity of ρ, we have that $\rho(M''_d) = O(\sum_{i=1}^{\lceil d \rceil} \rho(M'')) = O(R(N))$. Finally, because $e_{A,M,N} \le \lceil d \rceil e_{A'',M,N}$ we have that $M \prec M''_d$, so by monotonicity of ρ we have $\rho(M) \le \rho(M''_d) = O(R(N))$. □

4. MATRIX MULTIPLY: A TALE OF TWO ALGORITHMS

This section provides a concrete example of our approach by working through the analysis of two cache-oblivious matrix multiplication algorithms.

We analyze two variants of the cache-oblivious matrix multiplication algorithm of Frigo et al. [17], which we call MM-Inplace and MM-Scan(see Algorithms 1 and 2). Both algorithms divide each input matrix into four submatrices and perform eight recursive calls to compute submatrix products. Both run in $O(N^{3/2}/\sqrt{M}B)$ I/Os in the DAM model, which is optimal [20, 21].

Remarkably, only MM-inplace is optimally cache adaptive; MM-scan is a $\Theta(\log(N/B^2))$ factor away from being optimal in the cache-adaptive model. This shows that cache-oblivious algorithms are not always cache adaptive.

The two algorithms differ in how they combine the eight matrix sub-products. MM-Scan adds the eight matrix sub-products in one final linear scan, yielding a recurrence of $T(N) = 8T(N/4) + \Theta(1 + N/B)$ in DAM. MM-Inplace computes the eight matrix sub-products "in place," adding the results of elementary multiplications into the output matrix as soon as it computes them, and yielding a recurrence

Algorithm 1 Cache-oblivious matrix multiply of two $n \times n$ matrices (each of size $N = n^2$) with $\Theta(1 + N/B)$ linear scan [17]. In this pseudocode, A_{BR} refers to the Bottom Right quadrant of a matrix, A_{TL} the Top Left, etc.

```
function MM-Scan($n$,$A$,$B$)
    if $N = 1$ then
        return $A \times B$
    else
        $X_{TL} \leftarrow$ MM-Scan($n/2, A_{TL}, B_{TL}$)
        $X_{TR} \leftarrow$ MM-Scan($n/2, A_{TL}, B_{TR}$)
        $X_{BL} \leftarrow$ MM-Scan($n/2, A_{BL}, B_{TL}$)
        $X_{BR} \leftarrow$ MM-Scan($n/2, A_{BL}, B_{TR}$)
        $Y_{TL} \leftarrow$ MM-Scan($n/2, A_{TR}, B_{BL}$)
        $Y_{TR} \leftarrow$ MM-Scan($n/2, A_{TR}, B_{BR}$)
        $Y_{BL} \leftarrow$ MM-Scan($n/2, A_{BR}, B_{BL}$)
        $Y_{BR} \leftarrow$ MM-Scan($n/2, A_{BR}, B_{BR}$)

        $C \leftarrow X + Y$                                  ▷ Linear scan
        return $C$
    end if
end function
```

of $T(N) = 8T(N/4) + O(1)$ in the DAM model. Both algorithms require a tall cache of $\Theta(B^2)$ and both recurrences have the same solution in the DAM model.

However, the linear scans of MM-Scan are wasteful if they execute when memory is plentiful. For each input size N, we can construct a profile W^*_N that causes MM-Scan to run inefficiently by giving the algorithm lots of memory during its scans, when it can't use it, and very little memory at other times. As a result, the profile W^*_N will have a recursive structure that mimics that of MM-Scan: where MM-Scan performs a linear scan of size $\Theta(N)$, W^*_N will contain a square of size $\Theta(N)$. When MM-Scan performs a recursive call on a problem of size $N/4$, W^*_N will have a copy of $W^*_{N/4}$. Hence $\rho(W^*_N)$, the progress possible on W^*_N, will satisfy a recurrence relation very similar to the recurrence relation for MM-Scan. Where MM-Scan's recurrence relation has a term of the form $\Theta(N/B)$, corresponding to a linear scan of size N, $\rho(W^*_N)$'s recurrence will have a term of the form $\rho(\square_N)$. This will enable us to solve explicitly for $\rho(W^*_N)$, yielding a concrete counter-example to the optimality of MM-Scan.

Theorem 4.1. *The cache-oblivious matrix multiplication algorithm* MM-Scan *is a* $\Omega(\log(N/B^2))$ *factor away from being optimal when solving problem instances of size* N.

Proof. By Lemma 8.12, we have that $\rho_\mu(\square_X) = \Theta(X^{3/2})$ and $R_\mu(N) = \Theta(N^{3/2})$ constitute a progress bound for the naïve matrix multiplication problem.

MM-Scan is an $(8, 1/4, 1)$-regular algorithm. Hence, by Theorem 7.5 it is a $\Theta(\log_4(N/B^2))$-factor away from being optimally progressing.

Since MM-Inplace is optimally progressing, MM-Scan requires a speed augmentation of $\Theta(\log_4(N/B^2))$ to be able to beat MM-Inplace in solving problem instances of size N on all profiles. Hence, MM-Scan is a $\Theta(\log(N/B^2))$ factor away from being optimal. □

5. A GENERAL RECIPE FOR ANALYZING ALGORITHMS IN THE CA MODEL

We now explain how to generalize the construction from Section 4 to obtain a simple recipe for testing whether a recursive linear-space cache-oblivious algorithm is optimally progressing.

The method from Section 4 can be used to construct a profile W_N^* for any cache-oblivious algorithm. Whenever $\rho(W_N^*) = \omega(R(N))$, the algorithm is not optimal.

The main technical challenge of this paper is to show that W_N^* is, for many algorithms, the worst possible profile, up to constant factors. Thus, an algorithm is optimally progressing if and only if $\rho(W_N^*) = O(R(N))$.

Since $\rho(W_N^*)$ satisfies, by construction, a recurrence that we can easily derive from the program's structure, this gives us an easy way to analyze Master-method-style cache-oblivious algorithms in the CA model.

1. Write down the recurrence for $T(N)$, the algorithm's I/O complexity in the DAM model:
$$T(N) = aT(N/b) + \Theta(N^c/B).$$
2. Derive the recurrence for $\rho(W_N)$ by replacing terms of the form $T(X)$ with $\rho(W_X)$ and terms corresponding to linear scans of size X with $\rho(\square_X)$:
$$\begin{gathered} T(N) = aT(N/b) + \Theta(N^c/B) \\ \Downarrow \\ \rho(W_N) = a\rho(W_{N/b}) + \Theta(\rho(\square_{N^c})). \end{gathered}$$
3. Solve the recurrence for $\rho(W_N)$.
4. Compare the solution to $R(N)$. If $\rho(W_N) = O(R(N))$, then the algorithm is optimal. Otherwise, their ratio bounds how far the algorithm is from optimal.

In fact, we can explicitly solve the recurrence to obtain a simple theorem characterizing when linear-space complexity cache-oblivious Master-method-style algorithms are optimally progressing in the CA model (see Theorem 7.4).

The same basic technique works for Akra-Bazzi-style algorithms and even for collections of mutually-recursive Akra-Bazzi-style algorithms (see Theorem 6.12). Although Theorem 6.12 looks complex, the basic idea is the same.

1. Write down the recurrence relation for the I/O complexity of the algorithm.
2. Derive three new recurrences by performing the transformations specified in Theorem 6.12.
3. Solve these recurrences and compare to $R(N)$.

We explore these generalizations in Sections 6 and 7.

6. BOUNDING THE WORST-CASE PROFILE

This section formalizes the recipe described in Section 5. We first define the structure of algorithms covered by our theorems and then prove bounds on the progress possible on their worst-case profiles.

We first need to formalize the notion of a ***linear scan***. When a cache-efficient recursive algorithm is not making recursive calls, the work it does must be I/O efficient. We refer to this work as a ***linear scan***. Note that under our definition, a linear scan need not access a sequence of consecutive elements, as in a classical linear scan. However, it must be efficient—accessing $\Omega(B)$ useful locations on average, plus $O(1)$ additional I/Os.

Definition 6.1. *We say that algorithm L is a **linear scan of size ℓ** if it accesses ℓ distinct locations, it performs $\Theta(\ell)$ memory references, and its I/O complexity is $\Theta(1 + \ell/B)$.*

This definition captures a wide variety of efficient cache-oblivious behaviors. Note that, in the definition, a linear scan might not access every element of its input (e.g., a search for a specific item in an array), it might not access the pages in sequential order (e.g., cache-oblivious matrix transpose [17]), and the order of accesses can be data-dependent (e.g., the merge operation from merge-sort).

Note that some linear scans and algorithms are I/O efficient only under a tall-cache assumption. For example, for cache-oblivious sorting or matrix transpose, $H(B) = \Theta(B^2)$ [17]. Thus the definition of a scan depends implicitly on the memory profile.

A particular type of linear scans only access a small number of locations. This may occur when, for example, an algorithm does $O(1)$ work at the beginning of a recursive call to set up the computation, or when recursive calls decrease the size of the linear scan below the block size.

Definition 6.2. *When the size of a linear scan in an invocation of an algorithm is $\leq B$, we refer to it as an **overhead** reference. An overhead reference costs $\Theta(1)$ I/Os.*

We can now define the class of algorithms covered by our theorems. Briefly, these algorithms are collections of mutually recursive Akra-Bazzi-style algorithms, i.e. they make a constant number of recursive calls on sub-problems a constant factor smaller than their input, and perform linear scans. This class covers everything from simple algorithms, such as the matrix multiplication algorithms described earlier, to advanced cache-oblivious algorithms, such as the cache-oblivious longest-common-subsequence (LCS) and Edit Distance algorithms of Chowdhury and Ramachandran [11], and the cache-oblivious Jacobi Multipass Filter algorithm of Prokop [27].

Definition 6.3. *Let $0 \leq c_j \leq 1$ and $f_j \geq 1$ be constants for $j = 1, \ldots, e$. Also let $a_{ji} > 0$, and $0 < b_{ji} < 1$ be constants for $j = 1, \ldots, e$, and $i = 1, \ldots, f_j$. Algorithms $A_1, \ldots, A_e$ are **generalized compositional regular (GCR) algorithms** if, for all i, A_j on an input of size N*

(i) *makes $\boldsymbol{a_{ji}}$ calls to algorithm $\boldsymbol{A_{ji}}$ on subproblems of size $\boldsymbol{b_{ji}N}$. Algorithm A_{ji} is one of $A_1, \ldots, A_e$.*

(ii) *performs $\Theta(1)$ linear scans before, between, or after its calls, where the size of the biggest linear scan is $\Theta(N^{c_j})$.*

*Algorithms $A_1, \ldots, A_e$ are **perfect** generalized compositional regular (PGCR) algorithms, if for every j the size of* all *of A_j's linear scans is $\Theta(N^{c_j})$.*

We can now define the worst-case profile for an algorithm.

Definition 6.4. *Algorithm **A's worst-case profile for inputs of size N** among all profiles that are λ-tall is*

$$W_{A,N,\lambda} = \text{argmax}\{\rho(M) \mid M \textit{ is } N\textit{-fitting, } \lambda\textit{-tall, usable}\}.$$

When λ is omitted, we assume that λ equals the tall-cache requirement for A, $H(B)$,

$$W_{A,N} = W_{A,N,H} = \text{argmax}\{\rho(M) \mid M \textit{ is } N\textit{-fitting, } H\textit{-tall, usable}\}. \quad (1)$$

The following lemma, which follows directly from the definitions, enables us to analyze algorithms by looking at only their worst-case profiles.

Lemma 6.5. *If $\rho(W_{A,N,\lambda}) = O(R(N))$, then A is optimally progressing on all λ-tall profiles.*

When combined with Lemma 3.15, the above lemma means we can analyze algorithms by looking at only their worst-case square profiles.

The worst-case profile, or its inner square profile, does not have to respect the recursive structure of A. For example, squares can cross recursive boundaries, cover multiple recursive invocations, span multiple linear scans, etc. Any analysis based solely on the recursive structure of the algorithm must handle the fact that the profile may not nicely line up with the algorithm.

To solve this problem, we first establish a mapping from squares of any N-fitting square profile to recursive calls and linear scans performed by A.

The following definition defines three conditions under which the progress from a square can be charged to a linear scan, recursive call, or overhead reference.

Definition 6.6. *When A executes on a square profile $M(t)$, we say a square S of M **overlaps** a linear scan L if at least one memory reference of L is served during S. Similarly, we say S **encompasses** A's execution on a subproblem if every memory reference A makes while solving the subproblem is served during S. Finally, we say S **contains** an overhead reference R if at least half of the references of R are served during S.*

We now define when we can charge every square of a profile to a linear scan, recursive call, or overhead reference.

Definition 6.7. *Let $A_1, \ldots, A_e$ be generalized compositional regular (GCR) algorithms all with linear space complexity. We say that a square profile M of length ℓ is **N-chargeable with respect to A_j**, if every square S of M satisfies at least one of the following three properties when M is considered with respect to A_j's execution on any problem instance of size N that takes exactly ℓ steps to process.*

(i) S encompasses *an execution of any of $A_1, \ldots, A_e$ on a subproblem of size $\Theta(|S|)$.*

(ii) S overlaps *a linear scan of size $\Omega(|S|)$.*

(iii) S contains *$\Theta(|S|/B)$ overhead references.*

Finally, we prove that, for GCR algorithms with linear space complexity, every profile is N-chargeable.

We make use of the following notation throughout multiple proofs.

Notation 6.8. *The linear scans of an invocation of a* GCR *algorithm on an input of size x can be categorized as:*

- $\boldsymbol{L_1(x)}$*: $d_1 = \Theta(1)$ linear scans before any of the subcalls.*
- $\boldsymbol{L_{2,u}(x)}$*: $d_{2,u} = \Theta(1)$ linear scans between subcall u and subcall $u+1$.*
- $\boldsymbol{L_3(x)}$*: $d_3 = \Theta(1)$ linear scans at the end of all subcalls.*

Lemma 6.9. *Let e be a constant and let $A_1, \ldots, A_e$ be* perfect generalized compositional regular (PCGR) algorithms*, all with linear space complexity. Then every N-fitting square profile for A_j is N-chargeable with respect to A_j.*

Proof. Let M be an N-fitting profile for A_j and let S be a square of M and let σ be the sequence of memory references generated while solving a problem instance of size N for which M is I-fitting. We prove that S must have one of the three properties in Definition 6.7 with respect to σ.

Let N' be such that every $A_1, \ldots, A_e$ can solve problems of size less than or equal to N' using at most $|S|/3B$ I/Os and $|S|$ memory. Since every $A_1, \ldots, A_e$ has linear space complexity, $N' = \Theta(|S|)$.

Let $b = \min\{b_{ji}\} = \Theta(1)$. Note that every root-to-leaf path of the recursion tree must contain a subproblem whose size is in the range $[bN', N']$, so we can expand the recursion tree to subproblems of size between bN' and N'. Let $E_1, \ldots, E_t$ be the leaves of this partially expanded recursion tree, so that each E_i corresponds to an execution of an A_i on a problem of size between bN' and N'.

General properties of linear scans .

As before, we use Notation 6.8 to describe different types of linear scans in the recursive structure of A_j. Let Φ be any subsequence of memory references that does not contain a complete execution of any A_i. Thus Φ can contain only

- references generated by linear scans performed at the end of an execution of one of the A_is (L_3-type linear scans),
- references generated by an L_2-type linear scan between two recursive calls,
- references generated by linear scans performed at the beginning of an invocation of one of the A_is (L_1-type linear scans).

If Φ lies between two complete executions, then it may contain some number of L_3-type scans, followed by an L_2-type scan, followed by some number of L_1-type scans. If Φ consists of references in σ before the first complete execution of any A_i, then it contain references from only L_1-type scans. If Φ follows the last complete execution of any A_i in σ, then it will contain references from only L_3-type scans.

Property (i) .

If the square S encompasses an execution of any of $A_1, \ldots, A_e$ on a problem of size at least bN', then we are done, since $bN' = \Theta(N') = \Theta(|S|)$.

Properties (ii) and (iii) .

Suppose S does not encompass an invocation of A_j on a subproblem of size at least bN'. We show that S must either satisfy property (ii) or property (iii). In this case, S can intersect at most two of the leaves E_i and E_{i+1} of our partially expanded recursion tree (one at the beginning and one at the end). Furthermore, by the choice of N', these executions can occupy at most 2/3rds of the I/Os of S. Thus at least 1/3 the I/Os of S must be a contiguous sequence of memory references that does not contain a complete execution of any A_i. Call this subsequence Φ.

Let Z_1, Z_2, Z_3 be the set of linear scans of type L_1, $L_{2,u}$, and L_3, respectively, in Φ. Since S does not encompass a subproblem, the linear scans in Z_1 all belong to only one sequence of L_1-type *slide-down* moves on the recursion tree. Similarly, the linear scans in Z_3 all belong to only one sequence of L_3-type *climb-up* moves in the recursion tree.

Let $\mathcal{I}(.)$ denote the I/O complexity of a set of linear scans and allow $z = \max\{\mathcal{I}(Z_1), \mathcal{I}(Z_2), \mathcal{I}(Z_3)\}$. Since at least 1/3 of I/Os in S are overlapping linear scans, we have that $z \geq |S|/9B$. There are three cases to be considered.

Case of $z = \mathcal{I}(Z_2)$ In this case Z_2 is comprised of linear scans in only one $L_{2,u}$ set. Since $A_1, \ldots, A_e$ are all *perfect*, the size of all linear scans in $L_{2,u}$ is $\Theta(N^y)$ for some constant y.

If $y = 0$, then all scans in $L_{2,u}$ are overhead references and cost $\Theta(1)$ I/Os. Since, there are only $d_{2,u} = \Theta(1)$ linear scans in $L_{2,u}$, we deduce that $z = \mathcal{I}(Z_2) = \Theta(1)$. Since $z \geq |S|/9B$, S contains $d_{2,u} = \Theta(|S|/B)$ overhead references and thus satisfies property (iii).

If, otherwise $y > 0$, let $\mathcal{E}$ be the biggest linear scan in $L_{2,u}$. Because there are only $d_{2,u} = \Theta(1)$ linear scans in $L_{2,u}$, then $\mathcal{E}$ is a linear scan of size $\Omega(z) = \Omega(|S|)$. Thus, S would satisfy property (ii).

Case of $z = \mathcal{I}(Z_3)$ Here, Z_3 is comprised of linear scans of type L_3 from several invocations of (possibly) different A_j algorithms. Let L_3^j denote the set of all L_3 type linear scans in Z_3 that are executed in algorithm A_j's invocations.

Let L_3^m be the set among all L_3^js with the biggest I/O cost and let A_m be the algorithm that produced these scans. Since there are $e = \Theta(1)$ compositional algorithms, we have that $\mathcal{I}(L_3^m) \geq \mathcal{I}(Z_3)/e$. Let $x_1 \leq \cdots \leq x_k$ be the problem sizes solved by each of the invocations of A_m that generated one of the L_3-type linear scans in Z_3. Let $L_3^m(x_i)$ denote the linear scans generated by the invocation of A_m on problem of size x_m. Let $q = \max\{b_{ji}\}$ (note that $0 < q < 1$). Note that, since $A_1, \ldots, A_e$ are PGCR algorithms and the sequence of linear scans in L_3 come from invocations of $A_1, \ldots, A_j$ in *exactly one* sequence of climb-up moves, $x_i \leq q^{k-i} x_k$ for all i. Also, we have that

$$L_3^m = L_3^m(x_1) \cup L_3^m(x_2) \cdots \cup L_3^m(x_k).$$

If $k = 1$, then the analysis in case $(z = \mathcal{I}(Z_2))$ shows that S either satisfies property (ii) or property (iii).

So, we assume that $k > 1$. Since A_m is perfect, all linear scans in each $L_m^3(x_i)$ are all of size $\Theta(x_i^{c_m})$ for a constant c_m.

If $c_m = 0$, then all linear scans in L_m^3 are overhead references. Since each overhead reference costs $\Theta(1)$ I/Os and $\mathcal{I}(L_m^3) = \Omega(|S|/B)$ there must be $\Omega(|S|/B)$ overhead references in L_m^3 and S satisfies property (iii).

Otherwise, assume that $c_m > 0$. By definition, each invocation of A_m can only perform a constant number of L_3-type linear scans, so we can write $\mathcal{I}(L_m^3)$ as:

$$\begin{aligned}\mathcal{I}(L_m^3) &= \mathcal{I}(L_m^3(x_1)) + \mathcal{I}(L_m^3(x_2)) + \cdots + \mathcal{I}(L_m^3(x_k)) \\ &= \Theta\left(1 + \frac{x_1^{c_m}}{B}\right) + \cdots + \Theta\left(1 + \frac{x_k^{c_m}}{B}\right).\end{aligned}$$

Let v be the biggest index such that the size of the biggest linear scan in $L_m^3(t)$ is $\leq B$ for each $1 \leq t \leq v$. We compare

$$\sigma_1 = \mathcal{I}(L_m^3(x_1)) + \ldots \mathcal{I}(L_m^3(x_v))$$

and

$$\sigma_2 = \mathcal{I}(L_m^3(x_{v+1})) + \ldots \mathcal{I}(L_m^3(x_k)).$$

(a) If $\sigma_1 \geq \sigma_2$, we argue that S must satisfy property (iii). By definition of v, we have that all the linear scans in σ_1 are overhead references and cost $\Theta(1)$ I/Os. Since we have that $\sigma_1 \geq \mathcal{I}(L_m^3)/2 = \Omega(|S|/B)$, there must be $\Omega(|S|/B)$ overhead references in $L_m^3(x_1) \cup L_m^3(x_2) \ldots \cup L_m^3(x_v)$.

(b) If $\sigma_2 > \sigma_1$, we argue that S must satisfy property (ii). In this case, we show that $\mathcal{I}(L_m^3(x_k)) = \Omega(|S|/B)$ and hence $L_m^3(x_k)$ is a linear scan of size $\Omega(|S|)$ overlapped by S.

By definition of v, we have that the biggest linear scan in $L_m^3(x_{v+1})$ is of size $> B$. Because $x_{i+1} \geq x_i/q$ for each i. we have that the biggest linear scan in $L_m^3(x_t)$ has size bigger than $\Omega(B)$ for $v+1 \leq t \leq k$. Hence, we can write

$$\begin{aligned}\sigma_2 &= \mathcal{I}(L_m^3(x_{v+1})) + \cdots + \mathcal{I}(L_m^3(x_k)) \\ &= \Theta\left(\frac{x_{v+1}^{c_m}}{B}\right) + \cdots + \Theta\left(\frac{x_k^{c_m}}{B}\right).\end{aligned}$$

Consider the geometric series $\Psi = (x_k)^{c_m} + (qx_k)^{c_m} + \cdots + (q^{k-v}x_k)^{c_m}$ with constant coefficient $(q)^{c_m}$. Note that $\Psi \geq \sum_{t=v+1}^{k} x_t^{c_m}$, because $x_i \leq q^{k-i}x_k$. We have that

$$\begin{aligned}|L_m^3(x_k)| &= \Theta(x_k^{c_m}) = \Omega(\Psi) \\ &= \Omega\left(\sum_{t=v+1}^{k} x_t^{c_m}\right) = \Omega(B\sigma_2) = \Omega(|S|). \quad (2)\end{aligned}$$

Case of $z = \mathcal{I}(Z_1)$ The analysis in this case is identical to the case of $z = \mathcal{I}(Z_3)$.

We have established that each square S satisfies one of the three properties and the proof is complete. □

Finally, we bound the size of boxes that contain mostly overhead references, which means that they can be ignored when analyzing many algorithms.

Definition 6.10. *Let $A_1, \ldots, A_e$ be a set of generalized compositional regular algorithms. Let S be an overhead-containing square of a square N-fitting profile for A_j. For each* overhead reference *r in A_j, we let* **caller(r)** *be the call of any of $A_1, \ldots, A_e$ that created the reference r, and* ***rank(r)*** *be the input size of* caller(r). *Also we define*

$$\begin{aligned}\mathcal{E}(S) &= \{r | r \textit{ is an overhead reference contained in } S\}, \\ X(S) &= \text{caller}(\operatorname*{argmax}_{r \in \mathcal{E}(S)} \text{rank}(r)).\end{aligned}$$

$X(S)$ is the highest rank invocation of any of $A_1, \ldots, A_e$ that produced any of the overhead references in S.

Lemma 6.11. *Let $A_1, \ldots, A_e$ be a set of generalized compositional regular algorithms all with linear space complexity and let $q = \max\{b_{ji}\}$. Let M be an N-chargeable profile with respect to A_j. Each square of M that does not satisfy property (i) nor property (ii) in Definition 6.7 has size $O\left(B \log_{1/q} X(S)\right)$.*

Proof. Rob: This proof is written somewhat sloppily. Let S be a square of M that does not satisfy property (i) nor property (ii). Because

Theorem 6.12. *Let $0 \le c_j \le 1$ and $f_j \ge 1$ be constants for $j = 1, \ldots, e$. Also let $a_{ji} > 0$, and $0 < b_{ji} < 1$ be constants for $j = 1, \ldots, e$, and $i = 1, \ldots, f_j$. Suppose $A_1, \ldots, A_e$ are generalized compositional regular algorithms all with linear space complexity, tall-cache requirement $H(B)$, and progress bound ρ.*
Let $b = \max\{b_{ji}\}$ and $\lambda \ge H(B)$ be constants. Then there exist functions $\mathcal{T}_1, \ldots, \mathcal{T}_e$; $\mathcal{U}_1, \ldots, \mathcal{U}_e$; $\mathcal{V}_1, \ldots, \mathcal{V}_e$ such that the progress of the worst-case λ-tall profile for A_j, $\rho(W_{A_j,N,\lambda})$, is $O(\mathcal{T}_j(N) + \mathcal{U}_j(N) + \mathcal{V}_j(N))$ and the $\mathcal{T}_j$, $\mathcal{U}_j$ and $\mathcal{V}_j$ satisfy the recurrences

$$\mathcal{T}_j(N) = \begin{cases} \max\left\{\rho\left(\square_N\right), \displaystyle\sum_{i=1}^{f_j} a_{ji}\mathcal{T}_{ji}(b_{ji}N)\right\} & \text{if } \lambda < N \\ \Theta(\rho(\square_\lambda)) & \text{if } N \le \lambda; \end{cases}$$

$$\mathcal{U}_j(N) = \begin{cases} \Theta\left(\rho(\square_{N^{c_j}})\right) + \displaystyle\sum_{i=1}^{f_j} a_{ji}\mathcal{U}_{ji}(b_{ji}N) & \text{if } N = \Omega(\lambda) \text{ and } N^{c_j} = \Omega(\lambda) \\ \displaystyle\sum_{i=1}^{f_j} a_{ji}\mathcal{U}_{ji}(b_{ji}N) & \text{if } N = \Omega(\lambda) \text{ and } N^{c_j} \ne \Omega(\lambda) \\ 0 & \text{if } N \ne \Omega(\lambda); \end{cases}$$

$$\mathcal{V}_j(N) = \begin{cases} \displaystyle\sum_{i=1}^{f_j} a_{ji}\mathcal{V}_{ji}(b_{ji}N) & \text{if } B\log_{1/b} N = \Omega(\lambda) \text{ and } N^{c_j} > B \\ \Theta(\rho(\square_{B\log_{1/b} N})) + \displaystyle\sum_{i=1}^{f_j} a_{ji}\mathcal{V}_{ji}(b_{ji}N) & \text{if } B\log_{1/b} N = \Omega(\lambda) \text{ and } N^{c_j} \le B \\ 0 & \text{if } B\log_{1/b} N \ne \Omega(\lambda). \end{cases}$$

where $\mathcal{T}_{ji}$, $\mathcal{U}_{ji}$ and $\mathcal{V}_{ji}$ are one of $\mathcal{T}_1, \ldots, \mathcal{T}_e$; $\mathcal{U}_1, \ldots, \mathcal{U}_e$; $\mathcal{V}_1, \ldots, \mathcal{V}_e$ depending on the structure of A_j.

M is N-chargeable with respect to A_j, we have that S must contain $\Theta(|S|/B)$ overhead references.

Let $b = \min\{b_{ji}\} = \Theta(1)$, and expand the recursion to subproblems of size between bN' and N', where N' is chosen so that every $A_1, \ldots, A_e$ can solve problems of size less than or equal to N' using at most $|S|/3B$ I/Os and $|S|$ memory. Since every $A_1, \ldots, A_e$ has linear space complexity, $N' = \Theta(|S|)$.

S does not satisfy property (i), so it can not encompass a subproblem of size N'. The overhead references contained in S belong to the sequence of references between at most two invocations of any of $A_1, \ldots, A_e$ on subproblems of size N'.

$X(S)$ is the highest rank invocation of any of $A_1, \ldots, A_e$ that produced any of the overhead references in S. The size of the biggest subcall for any $A_1, \ldots, A_e$ on a subproblem of size $X(S)$ is $qX(S)$. This means that between any two invocations of any of $A_1, \ldots, A_e$ on two subproblems that are overlapped by S there are at most $O(\log_{1/q} X(S))$ linear scans, and consequently, overhead references.

Since S must contain $\Theta(|S|/B)$ overhead references, and there can be at most $O(\log_{1/q} X(S))$ overhead references contained in S, we have we have that $|S_1| = O(B\log_{1/q} X(S))$. □

These two lemmas give us the "recurrence-rewriting" rule outlined in Section 5. The progress on each square of a profile can be charged to either (1) a linear scan of size at least as large as the square, (2) a subproblem of size proportional to the square, or (3) overhead references, although in the last case the square cannot be very large. Theorem 6.12 (at the top of the page) summarizes this result.

Proof of Theorem 6.12. Lemma 6.9 shows that all N-fitting profiles for *perfect* generalized compositional regular algorithms are N-chargeable. However, algorithms $A_1, \ldots, A_j$ are not necessary perfect (according to Definition 6.3), so Lemma 6.9 does not apply to them.

We modify $A_1, \ldots, A_j$ to get ***padded algorithms*** $\boldsymbol{A'_1, \ldots, A'_e}$ and exhibit that the padded algorithms are PGCR algorithms. Each A'_j operates exactly in the same way as A_j, except that on in each invocation of size Y, A'_j ***pads*** all linear scans of A_j to be as big as the biggest linear scan in A_j's invocation of size Y, $\Theta(Y^{c_j})$. The *padding* operation can be done by scanning an auxiliary array of appropriate size.

We argue that $\rho(W_{A_j,N,\lambda}) \le \rho(W_{A'_j,N,\lambda})$. We take $W_{A_j,N,\lambda}$ as a profile and modify it to get a profile Φ_j and show that Φ_j is N-fitting for A'_j. Consider $W_{A_j,N,\lambda}$ and in a bottom-up manner extend squares of $W_{A_j,N,\lambda}$ so that these squares are big enough to serve all *padded* linear scans in A'_j. Since $W_{A_j,N,\lambda}$ is N-fitting for A_j, by definition Z'_j is N-fitting for A'_j. By Definition 3.6 we have that $W_{A_j,N,\lambda} \prec \Phi_j$. Hence, by Definition 3.7

$$\rho(W_{A_j,N,\lambda}) \le \rho(\Phi_j) \le \rho(W_{A'_j,N,\lambda}). \qquad (3)$$

Let M_j be any λ-tall N-fitting profile for A'. We bound the progress of M_j thus bound $\rho(W_{A'_j,N,\lambda})$ from above. Lemma 6.9 shows that M_j is N-chargeable.

If $N \le \lambda$, because A'_j is PCGR and has linear space complexity, if one unrolls the recursion of A'_j a constant number of times, whole executions of subproblems can be completed inside a single square of size λ. It follows that M_j must consist of $\Theta(1)$ squares of size λ, so $\rho(M_j) = \Theta(\rho(\square_\lambda)) = \Theta(\mathcal{T}_j(N))$ for any $j = 1, \ldots, e$.

Now, let $N > \lambda$. Because M_j is N-chargeable, every square S in M_j satisfies one of the three properties in Definition 6.7.

Charging the subproblem-encompassing squares First, we will charge the progress of each subproblem-encompassing square to the covered subproblem. When a square S is charged to a subproblem Z, all subproblems of Z are encompassed by S. Because A'_j has linear space complexity, S has size $\Theta(|Z|)$. Therefore, the progress of all squares charged to subproblems is bounded by $\Theta(\mathcal{T}_j(N))$ where $\mathcal{T}_j(N)$ satisfies the recurrence

$$\mathcal{T}_j(N) = \begin{cases} \max\left\{\rho\left(\square_N\right), \sum_{i=1}^{f_j} a_{ji}\mathcal{T}_{ji}(b_{ji}N)\right\} & \text{if } \lambda < N \\ \Theta(\rho(\square_\lambda)) & \text{if } N \le \lambda. \end{cases}$$

Charging the linear-scan-overlapping squares If a square S overlaps a linear scan L of size $\Omega(|S|)$ executed by the top-level invocation of A'_j and L is the *biggest* linear scan overlapped by S, we charge it to L. At the top-level invocation of A'_j, all linear scans are of size $\Theta(N^{c_j})$.

We develop a recursive relation $\mathcal{U}_j(N)$ that bounds the total progress of all linear-scan-overlapping squares for an invocation of A'_j on a problem instance of size N.

Let N_0 be the size of input in an invocation for A'_j. If $N_0 \ne \Omega(\lambda)$, then because M_j is λ-tall, no square of M_j can be charged to a linear scan executed in any part of the subproblem $A'_j(N_0)$, because squares are much bigger than these linear scans. Therefore, $\mathcal{U}_j(N_0) = 0$.

Now consider an input size of $N_1 = \Omega(\lambda)$ for A'_j. If $N_1^{c_j} \ne \Omega(\lambda)$, then because M_j is λ-tall, no square in M_j can be charged to a linear scan executed in the top level invocation of $A'_j(N_1)$. However, other subcalls of A'_j might execute linear scans that are large enough. Therefore, $\mathcal{U}_j(N_1) = \sum_{i=1}^{f_j} a_{ji}\mathcal{U}_{ji}(b_{ji}N_1)$.

For larger N, multiple squares may have their progress charged to a single linear scan of size $|L|$, but all but at most two of those squares will be contained entirely within the linear scan. Thus, the total size of all the squares charged to a single linear scan of size $|L|$ will be $\Theta(|L|)$. Suppose $S_1, \ldots, S_k$ are the squares charged to L. Since $\rho(\square_X) = \Omega(X)$, we must have that $\sum \rho(S_i) = O(\rho(\square_{\sum|S_i|})) = O(\rho(\square_{|L|}))$. Thus the progress of all the squares charged to linear scans executed by the top-level invocation of A'_j on a problem instance of size N can be upper-bounded by $\Theta\left(\rho(\square_{N^{c_j}})\right)$.

Therefore, the progress of all squares charged to linear scans is upperbounded by $\mathcal{U}_j(N)$

$$\mathcal{U}_j(N) = \begin{cases} \Theta\left(\rho(\square_{N^{c_j}})\right) + \sum_{i=1}^{f_j} a_{ji}\mathcal{U}_{ji}(b_{ji}N) \\ \qquad\qquad \text{if } N = \Omega(\lambda) \text{ and } N^{c_j} = \Omega(\lambda) \\ \sum_{i=1}^{f_j} a_{ji}\mathcal{U}_{ji}(b_{ji}N) \\ \qquad\qquad \text{if } N = \Omega(\lambda) \text{ and } N^{c_j} \ne \Omega(\lambda) \\ 0 \qquad\qquad \text{if } N \ne \Omega(\lambda); \end{cases}$$

Charging the overhead-containing squares Finally, we will charge each overhead-containing square S_1 to the recursive call that corresponds to the subproblem $X(S_1)$, *the highest rank subproblem that produced any of the overhead references in* S_1 (see Definition 6.10). By Lemma 6.11, the size of each overhead-containing square, S_1, of M_j that is not subproblem-encompassing nor linear-scan-overlapping is $O(B\log_{1/b} X(S_1))$.

If $c_j = 0$, then all linear scans of A'_j are overhead references. Otherwise, A'_j's linear scan *convert* to overhead references when $N^{c_j} \le B$.

We develop a recursive relation $\mathcal{V}_j(N)$ that bounds the total progress of all overhead-containing squares for an invocation of A'_j on a problem instance of size N. At the top-level invocation of A'_j of size N, we only account for overhead-containing squares whose highest rank overhead are produced at the current invocation. Hence, their size is $O(B\log_{1/b} N)$.

Let N_3 be the size of input in an invocation for A'_j. If $B\log_{1/b} N_3 \ne \Omega(\lambda)$, then because M_j is λ-tall, no square of M_j can be charged to the recursive call $A'_j(N_3)$, because squares are much bigger than a series of overhead references whose highest ranked reference is executed in $A'_j(N_3)$. Therefore, $\mathcal{V}_j(N_3) = 0$.

Now consider N_4 to be an input size for A'_j, such that $B\log_{1/b} N_4 = \Omega(\lambda)$. If $N_4^{c_j} \ne O(B)$, then the linear scans in the top-level invocation of A'_j are *not* overhead references. However, other subcalls of A'_j might execute overhead references. Hence, $\mathcal{V}_j(N_4) = \sum_{i=1}^{f_j} a_{ji}\mathcal{V}_{ji}(b_{ji}N_4)$.

Since each invocation of A'_j executes $\Theta(1)$ overhead references, the progress of all overhead-containing squares that contain an overhead executed by the top-level invocation of A'_j can be upper-bounded by $\Theta(\rho(\square_{B\log_{1/b} N}))$. Therefore, the progress of all overhead-containing squares is upperbounded by $\mathcal{V}_j(N)$

$$\mathcal{V}_j(N) = \begin{cases} \sum_{i=1}^{f_j} a_{ji}\mathcal{V}_{ji}(b_{ji}N) \\ \qquad\qquad \text{if } B\log_{1/b} N = \Omega(\lambda) \text{ and } N^{c_j} > B \\ \Theta(\rho(\square_{B\log_{1/b} N})) + \sum_{i=1}^{f_j} a_{ji}\mathcal{V}_{ji}(b_{ji}N) \\ \qquad\qquad \text{if } B\log_{1/b} N = \Omega(\lambda) \text{ and } N^{c_j} \le B \\ 0 \qquad\qquad \text{if } B\log_{1/b} N \ne \Omega(\lambda). \end{cases}$$

We have charged squares of every type in M_j. Therefore, we have that $\rho(M_j) = O(\mathcal{T}_j(N) + \mathcal{U}_j(N) + \mathcal{V}_j(N))$.

Since, M_j can be any N-fitting profile for A'_j, we have shown that

$$\begin{aligned} \rho(W_{A_j,N,\lambda}) &\le \rho(W_{A'_j,N,\lambda}) && \text{by Eq. 3,} \\ &\le O(\mathcal{T}_j(N) + \mathcal{U}_j(N) + \mathcal{V}_j(N)), \end{aligned}$$

which finishes the proof of the theorem. □

The following theorem gives a corresponding lower bound on the progress of the worst-case profile. While Theorem 6.12 can show that an algorithm is optimally progressing, Theorem 6.13 gives a recipe to prove that an algorithm is not optimally progressing.

Theorem 6.13. *Suppose $A_1, \ldots, A_e$ are generalized compositional regular algorithms with linear space complexity, tall-cache requirement $H(B)$, and progress bound ρ. Let $\lambda = \max\{H(B), (B\log_{1/b} B)^{1+\varepsilon}\}$, where ε is any constant larger than 0. When all $c_j = 1$, the bound in Theorem 6.12 is tight, meaning that $\rho(W_{A_j,N,\lambda})$, is $\Theta(\mathcal{T}_j(N) + \mathcal{U}_j(N) + \mathcal{V}_j(N))$.*

Proof. We describe how to build an N-fitting profile for A_j, M_j, such that $\rho(M_j) = \Theta(\mathcal{T}_j(N) + \mathcal{U}_j(N) + \mathcal{V}_j(N))$.

We first prove that when $c_j = 1$, and M_j is λ-tall we have that for all j, $\mathcal{V}_j(N) = 0$. Note that when $N^1 \leq B$, $B \log_{1/b} N \leq B \log_{1/b} B$. Because $\lambda \geq (B \log_{1/b} B)^{1+\varepsilon}$, we have that $B \log_{1/b} N \neq \Omega(\lambda)$. Therefore the middle case in $\mathcal{V}_j(N)$ never happens for any of $\mathcal{V}_j(N)$s. Thus for all j, $\mathcal{V}_j(N) = 0$.

Now consider $\Theta(\mathcal{T}_j(N)+\mathcal{U}_j(N))$ and among the two terms in the Θ, pick the term that is bigger.

First, assume that $\mathcal{U}_j(N)$ is bigger. We construct an N-fitting profile W_N such that $\rho(W_N) = \Theta(U_j(N))$ as follows. Memory starts at λ. Whenever the algorithm begins a linear scan of size $L \geq 2\lambda$ with memory size λ, it will necessarily incur at least $(L - \lambda)/B = \Theta(L/B)$ cache misses, no matter how memory changes size during the linear scan. Thus we can increase the size of memory to $O(L)$ at the first page fault during the linear scan and decrease it to size λ on the last page fault during the scan, which will be $\Theta(L/B)$ time steps later. Thus W_N will contain a square of size $\Theta(L)$ for every linear scan of size $L = \Omega(\lambda)$ performed during the execution of A_j on a problem of size N. Since each A_i performs linear scans of size N^{c_j} on inputs of size N, $\rho(W_N) = \Omega(\mathcal{U}_j(N))$.

Now assume that $\mathcal{T}_j(N)$ is bigger.

We first construct a slightly different recurrence on functions $\mathcal{T'}_j(N)$, show that $\mathcal{T'}_j(N) = \Theta(\mathcal{T}_j(N))$, and then construct a profile W_N such that $\rho(W_N) = \Theta(\mathcal{T'}_j(N))$.

Let $s_j(N)$ be the space complexity of A_j on problems of size N. Let $b = \min\{b_{ji}\}$. Let

$$\mathcal{T'}_j(N) = \begin{cases} \max\left\{ \rho(\square_N, \displaystyle\sum_{i=1}^{f_j} a_{ji} \mathcal{T'}_{ji}(b_{ji}N) \right\} & \text{if } s_j(N) \geq 2\lambda/b \\ \Theta(\rho(\square_\lambda) & \text{otherwise.} \end{cases}$$

Since $\mathcal{T}_j$ and $\mathcal{T'}_j$ differ only in a constant factor of the sizes of their base cases, $\mathcal{T'}_j(N) = \Theta(\mathcal{T}_j(N))$.

$\mathcal{T'}_j(N)$ is the max among a finite number of terms. Take any term, Φ, that is maximal, i.e. $\mathcal{T'}_j(N) = \Phi$. Because $\mathcal{T'}_j(N)$ is based on the recursive structure of A_j, Φ is the sum of terms $\rho(\square_{n_1}) + \cdots + \rho(\square_{n_t})$ corresponding to different subproblems of $A_1, \ldots, A_e$s of sizes $n_1, \ldots, n_t$ such that no two subproblems are included in one another. Note that, for each of these problems, their space complexity is at least 2λ, by the definition of $\mathcal{T'}_j$.

Given these non-overlapping subproblems of size $n_1, \ldots, n_t$, we construct W_N as follows. Memory starts out at size λ. Whenever the algorithm begins solving a problem of size n_i with λ memory, it will necessarily incur $\Theta(n_j/B)$ page faults because the problem's space complexity is linear (and at least 2λ by definition of $\mathcal{T'}_j(N)$). Thus we can increase memory to size $\Theta(n_j)$ on the first page fault during the algorithms execution on the problem, and decrease it to size λ on the last page fault during the algorithm's execution on the problem, which must be at least $\Theta(n_j/B)$ time steps later. Thus $\rho(W_N) = \Theta(\rho(\square_{n_1}) + \cdots + \rho(\square_{n_t})) = \Theta(\mathcal{T'}_j(N)) = \Theta(\mathcal{T}_j(N))$. □

7. OPTIMALITY CRITERIA FOR MANY MASTER-METHOD-STYLE ALGORITHMS

Theorem 6.12, Theorem 6.13, and Lemma 6.5 provide an easy way to test whether a linear-space GCR algorithm is optimally progressing: derive the recurrences on $\mathcal{T}_i$, $\mathcal{U}_i$, and $\mathcal{V}_i$ from the structure of the algorithm, solve the recurrences, and check whether $\mathcal{T}_i(N) + \mathcal{U}_i(N) + \mathcal{V}_i(N) = O(R(N))$.

Although we can't give a general solution for all possible $\mathcal{T}_i$, $\mathcal{U}_i$, and $\mathcal{V}_i$, we can use this method to derive a simple test for Master-method-style algorithms.

We first define the class of Master-theorem-style algorithms covered by our optimality criterion. Note that this class is more general than the constant-overhead recursive (COR) form algorithms covered by previous work [8]. Note also that $c \leq 1$ is implied by the linear space restriction.

Definition 7.1. *Let $a \geq 1/b$, $0 < b < 1$, and $0 \leq c \leq 1$ be constants. A linear-space algorithm is* ***(a, b, c)-regular*** *if, for inputs of sufficiently large size N, it makes*

(i) *exactly* ***a*** *recursive calls on subproblems of size* ***bN****, and*

(ii) $\Theta(1)$ *linear scans before, between, or after recursive calls, where the size of the biggest scan is $\Theta(N^c)$.*

We will prove that a DAM-optimal linear-space-complexity (a, b, c)-regular algorithm is optimal in the CA model if and only if $c < 1$. We first solve the recurrence from Theorem 6.12 for the special case of (a,b,c)-regular algorithms to get a simple bound on $\rho(W_N)$. The proof depends on the following lemma.

Lemma 7.2. *Assume that $B \geq 4$. Pick a $\delta \in (0, 0.1)$, and let $d = 3(1 + \delta)$. If Z is bigger than $(dB \log B)^{1+\delta}$, we have that $Z^{1/(1+\delta)} > B \log Z$.*

Proof. Because $\varepsilon \in (0, 0.1)$ we have that $d = 3(1 + \delta) < 3.3 < B$.

Second, notice that as long as $\delta \in (0, 0.1)$, $f(x) = x^{1/(1+\delta)}/\log x$ is a strictly increasing function in x on $[e, \infty)$. Let $x_0 = (dB \log B)^{1+\delta}$, and observe that

$$\log x_0 = 3(1 + \delta)(\log d + \log B + \log \log B) < 3(1 + \delta) \log B.$$

Therefore,

$$\frac{x_0^{1/(1+\delta)}}{\log x_0} = \frac{dB \log B}{\log x_0} > B \qquad \text{since } d = 3(1 + \delta).$$

Since $x^{1/(1+\delta)}/\log x$ is strictly increasing on $[e, \infty)$ and $Z > (dB \log B)^{1+\delta}$, we get that $Z^{1/(1+\delta)} > B \log Z$. □

The next lemma shows that, interestingly, if an optimal algorithm has a tight progress bound, then (up to constants), the problem has only one possible progress bound. This lemma is the crucial building block in proving our tight characterization of (a, b, c)-regular algorithms, as it allows us to state ρ specifically.

Lemma 7.3. *Let A be an (a, b, c)-regular algorithm with linear space complexity and tall-cache requirement $H(B)$. Suppose A is optimal in the DAM model for a problem with progress bound $\rho(\square_X) = \Theta(X^p)$. Then, $p = \log_{1/b} a$.*

Proof. Let $M_1 = H(B)$ and let $M_2 \geq H(B)$ be an arbitrary constant. Since (a, b, c)-regular algorithms are cache-oblivious, A is optimal in all DAM memory profiles where the amount of memory is bigger than $H(B)$, Specifically, A is optimal on both profiles M_1 and M_2.

Let $e_{A,M_1,N}$ be the minimum time required for A to solve all instances of size N given M_1, and $e_{A,M_2,N}$ the equivalent time with respect to M_2. Define

$$M'_1(t) = \begin{cases} M_1 & \text{if } t \le e_{A,M_1,N}, \\ 0 & \text{otherwise;} \end{cases}$$

$$M'_2(t) = \begin{cases} M_2 & \text{if } t \le e_{A,M_2,N}, \\ 0 & \text{otherwise.} \end{cases}$$

We compute $e_{A,M_1,N}$ and $e_{A,M_2,N}$

$$e_{A,M_1,N} = \Theta\left(\frac{M_1}{B} a^{\log_{1/b} N/M_1}\right) = \Theta\left(\frac{M_1}{B}\left(\frac{N}{M_1}\right)^{\log_{1/b} a}\right)$$

$$e_{A,M_2,N} = \Theta\left(\frac{M_2}{B} a^{\log_{1/b} N/M_2}\right) = \Theta\left(\frac{M_2}{B}\left(\frac{N}{M_2}\right)^{\log_{1/b} a}\right).$$

Since ρ is square-additive, we can compute the progress of ρ on M'_1 and M'_2 as the sum of ρ on squares $\square_{M_1}$ and $\square_{M_2}$ respectively. Remember that each square $\square_X$ is X words tall and X/B time steps wide. Hence, M'_1 fits between $\left\lceil \frac{e_{A,M_1,B}}{M_1/B} \right\rceil - 1$ and $\left\lceil \frac{e_{A,M_1,B}}{M_1/B} \right\rceil$ of $\square_{M_1}$s. Similarly, M'_2 fits between $\left\lceil \frac{e_{A,M_2,B}}{M_2/B} \right\rceil - 1$ and $\left\lceil \frac{e_{A,M_2,B}}{M_2/B} \right\rceil$ of $\square_{M_2}$s.

$$\begin{aligned} \rho(M'_1) &= \Theta\left(\sum_{i=1}^{\frac{e_{A,M_1,B}}{M_1/B}} \rho(\square_{M_1})\right) = \Theta\left(\frac{e_{A,M_1,B}}{M_1/B}\rho(\square_{M_1})\right) \\ &= \Theta\left(\left(\frac{N}{M_1}\right)^{\log_{1/b} a} \rho(\square_{M_1})\right) \\ \rho(M'_2) &= \Theta\left(\sum_{i=1}^{\frac{e_{A,M_2,B}}{M_2/B}} \rho(\square_{M_2})\right) = \Theta\left(\frac{e_{A,M_2,B}}{M_2/B}\rho(\square_{M_2})\right) \\ &= \Theta\left(\left(\frac{N}{M_2}\right)^{\log_{1/b} a} \rho(\square_{M_2})\right). \end{aligned}$$

By Lemma 3.18, we have that A is optimally progressing on both M_1 and M_2. Therefore, $\rho(M'_1) = O(R(N))$ and $\rho(M'_2) = O(R(N))$. Since M'_1 and M'_2 are N-fitting profiles for A, we get that $\rho(M'_1) = \Theta(R(N))$ and $\rho(M'_2) = \Theta(R(N))$.

Therefore, $\rho(M'_1) = \Theta(\rho(M'_2))$ and we get

$$\begin{aligned} \rho(M'_1) &= \Theta\left(\left(\frac{N}{M_1}\right)^{\log_{1/b} a} \rho(\square_{M_1})\right) \\ &= \Theta\left(\left(\frac{N}{M_2}\right)^{\log_{1/b} a} \rho(\square_{M_2})\right) = \rho(M'_2) \end{aligned} \tag{4}$$

By simplifying Equation (4), we get that

$$\begin{aligned} \rho(\square_{M_2}) &= \Theta\left(\left(\frac{M_2}{M_1}\right)^{\log_{1/b} a} \rho(\square_{M_1})\right) \\ &= \Theta\left(M_2^{\log_{1/b} a} \frac{\rho(\square_{M_1})}{(M_1)^{\log_{1/b} a}}\right). \end{aligned} \tag{5}$$

Since M_2 is an arbitrary constant, we can allow $M_2 = M_1^2$ and then we will have

$$\rho(\square_{M_1^2}) = \Theta\left(M_1^{\log_{1/b} a} \rho(\square_{M_1})\right).$$

Since $\rho(\square_X) = \Theta(X^p)$, we have that $\rho(\square_{X^2}) = \Theta(\rho(\square_X)^2)$. Therefore,

$$\begin{aligned} \rho(\square_{M_1^2}) &= \Theta\left(M_1^{\log_{1/b} a} \rho(\square_{M_1})\right) = \Theta\left(\rho(\square_{M_1})^2\right) \\ &\Rightarrow \rho(\square_{M_1}) = \Theta\left(M_1^{\log_{1/b} a}\right). \end{aligned} \tag{6}$$

Therefore, we have shown that for an arbitrary M_2

$$\rho(\square_{M_2}) = \Theta\left(M_2^{\log_{1/b} a} \frac{\rho(\square_{M_1})}{(M_1)^{\log_{1/b} a}}\right) = \Theta\left(M_2^{\log_{1/b} a}\right).$$

□

Using these two results, we are ready to prove the characterization theorem.

Theorem 7.4. *Let A be an (a,b,c)-regular algorithm with linear space complexity and tall-cache requirement $H(B)$. Suppose that A is optimal in the DAM model for a problem with progress bound $\rho(\square_X) = \Theta(X^p)$, where p is a constant. Assume that $B \ge 4$. Pick an $\varepsilon > 0$, and let $d = 3(1+\varepsilon)$ and $\lambda = \max\{H(B), (dB\log_{1/b} B)^{1+\varepsilon}\}$.*

Then, $\rho(W_{A,N,\lambda})$ is bounded by $O(\mathcal{X}(N))$, where

$$\mathcal{X}(N) = \begin{cases} \Theta\left(N^{\log_{1/b} a} \log_{1/b} \frac{N}{\lambda}\right) & \textit{if } c = 1 \\ \Theta\left(N^{\log_{1/b} a}\right) & \textit{otherwise.} \end{cases}$$

Proof. By Lemma 7.3, we have that because A is optimal in the DAM model, $p = \log_{1/b} a$. By Theorem 6.12, $\rho(W_{A,N,\lambda}) = O(\mathcal{T}(N) + \mathcal{U}(N) + \mathcal{V}(N))$. where

$$\mathcal{T}(N) = \begin{cases} \max\left\{\Theta\left(N^{\log_{1/b} a}\right), a\mathcal{T}(bN)\right\} & \text{if } \lambda < N \\ \Theta\left(\lambda^{\log_{1/b} a}\right) & \text{if } N \le \lambda; \end{cases}$$

$$\mathcal{U}(N) = \begin{cases} \Theta\left(N^{c\log_{1/b} a}\right) + a\mathcal{U}(bN) & \\ \qquad \text{if } N = \Omega(\lambda) \text{ and } N^c = \Omega(\lambda) \\ a\mathcal{U}(bN) & \\ \qquad \text{if } N = \Omega(\lambda) \text{ and } N^c \ne \Omega(\lambda) \\ 0 & \text{otherwise;} \end{cases}$$

$$\mathcal{V}(N) = \begin{cases} a\mathcal{V}(bN) & \\ \qquad \text{if } B\log_{1/b} N = \Omega(\lambda) \text{ and } N^c > B \\ \Theta\left((B\log_{1/b} N)^{\log_{1/b} a}\right) + a\mathcal{V}(bN) & \\ \qquad \text{if } B\log_{1/b} N = \Omega(\lambda) \text{ and } N^c \le B \\ 0 & \text{if } B\log_{1/b} N \ne \Omega(\lambda). \end{cases}$$

Solving the recursion for $\mathcal{T}(N)$ using the Master method we get

$$\mathcal{T}(N) = \Theta\left(N^{\log_{1/b} a}\right).$$

As for $\mathcal{U}(N)$, we note that $a(bN)^{c\log_{1/b} a} = a^{1-c}N^{c\log_{1/b} a}$. When $0 < c < 1$, we have that $\mathcal{U}(N)$ becomes a geometric series and solves to $\Theta\left(N^{c\log_{1/b} a}\right)$. When $c = 1$, we have that $\mathcal{U}(N)$ is the summation of $\log_{1/b} N/\lambda$ terms, each of them equal to $\Theta\left(N^{\log_{1/b} a}\right)$. And when $c = 0$, $N^c \neq \Omega(\lambda)$. Hence,

$$\mathcal{U}(N) = \begin{cases} \Theta\left(N^{\log_{1/b} a} \log_{1/b} \frac{N}{\lambda}\right) & \text{if } c = 1 \\ 0 & \text{if } c = 0 \\ \Theta\left(N^{c \log_{1/b} a}\right) & \text{otherwise.} \end{cases}$$

We bound $\mathcal{V}(N)$ from above. Note that as long as $N_0 > \lambda \geq (dB \log_{1/b} B)^{1+\varepsilon}$, by Lemma 7.2 we have that

$$\frac{N_0^{1/(1+\varepsilon)}}{\log_{1/b} N_0} > B \Rightarrow N_0^{1/(1+\varepsilon)} > B \log_{1/b} N_0.$$

And when $N_1 \leq \lambda$, we have $B \log_{1/b} N_1 \leq B \log_{1/b} \lambda < \lambda^{1/(1+\varepsilon)} < \lambda$ by another application of Lemma 7.2 because $\lambda \geq (dB \log_{1/b} B)^{1+\varepsilon}$. Therefore $\mathcal{V}(N) = O(\mathcal{Z}(N))$ where

$$\mathcal{Z}(N) = \begin{cases} \Theta\left(N^{\frac{\log_{1/b} a}{1+\varepsilon}}\right) + a\mathcal{Z}(bN) & \text{if } N > \lambda \\ \Theta\left(\lambda^{\log_{1/b} a}\right) & \text{if } N \leq \lambda. \end{cases}$$

$$= O\left(N^{\log_{1/b} a}\right) \qquad \text{by applying the Master method.}$$

The theorem statement follows from summing the terms for $\mathcal{T}(N)$, $\mathcal{U}(N)$ and $\mathcal{V}(N)$. □

The following rule for optimality of (a, b, c)-regular algorithms can be derived by comparing $\mathcal{X}(N)$ with $R(N)$.

Theorem 7.5. *Suppose A is an (a, b, c)-regular algorithm with tall-cache requirement $H(B)$ and linear space complexity. Suppose also that, in the DAM model, A is optimally progressing for a problem with progress bound $\rho(\square_N) = \Theta(N^p)$, for constant p. Assume $B \geq 4$. Let $\lambda = \max\{H(B), ((1+\varepsilon)B \log_{1/b} B)^{1+\varepsilon)}\}$, where $\varepsilon > 0$.*

1. *If $c < 1$, then A is optimally progressing and optimally cache-adaptive among all λ-tall profiles.*
2. *If $c = 1$, then A is $\Theta\left(\log_{1/b} \frac{N}{\lambda}\right)$ away from being optimally progressing and $O\left(\log_{1/b} \frac{N}{\lambda}\right)$ away from being optimally cache-adaptive.*

Proof. Suppose M is a square N-fitting profile for A.

By Lemma 7.3, we have that $p = \log_{1/b} a$. If $c < 1$, then by Theorem 7.4 the maximum possible progress that any algorithm can make on M is $\rho(M) = \Theta(N^{\log_{1/b} a})$. Since A is DAM-optimal and has linear space complexity, $R(N) = \Theta(\rho(\square_N)) = \Theta(N^{\log_{1/b} a})$. Thus $\rho(M) = \Theta(R(N))$.

If $c = 1$, then by Theorem 7.4 and Theorem 6.13 A is a factor of $\Theta(\log_{1/b} N/\lambda)$ away from being optimally progressing. Hence, with $O(\log_{1/b} N/\lambda)$ speed augmentation, A can out outperform any other memory-monotone algorithm. □

8. DERIVING PROGRESS BOUNDS

In this section, we exhibit that one can derive progress bounds, which satisfy the axioms of Definition 3.10, for several important problems. This derivation allows us to apply the general results of Section 7 to determine the cache-adaptivity of algorithms which solve these problems.

We utilize the powerful *red-blue pebble game* technique of Hong and Kung [20] and a similar *red pebble game* technique of Savage [28] to define appropriate functions that satisfy the axioms of Definition 3.10.

We begin by describing a computation DAG [20,28] which is an abstract way of describing computational dependencies of algorithms.

Definition 8.1. *Computational dependencies of an algorithm A can be described by a **computation DAG, G**. G is comprised of input/output nodes that have no incoming/outgoing edges. These nodes correspond to the input/output of the problem.*

Each internal *node of G represents a computation step by the algorithm. Node u is connected to node v via a directed edge $(u \to v)$, if computing node v requires data/information from node u.*

We adopt a notion of progress that bears similarity to the *red pebble game* approach of [28]. We define R and ρ with respect to DAGs of computation.

Definition 8.2. *Given a DAG of computation, G, for a problem instance of size N we define $\boldsymbol{R(N)}$ to be the number of nodes in G.*

We let $\boldsymbol{\rho(\square_S)}$ to be the maximum number of nodes of G computable using a fixed memory of size S and S/B I/Os, maximized over all initial memory contents, and also maximized over all instances.

Similarly, for a memory profile M, we allow $\boldsymbol{\rho(M)}$ to be the maximum number of nodes of G computable using profile M, maximized over all initial memory contents, and also maximized over all instances.

We can also define R and ρ for families of DAGs.

Definition 8.3. *For a family of DAGs, $\{\mathcal{G}_i\}$, such that the number of nodes in all $\mathcal{G}_i$s is the same, we define $\boldsymbol{\rho(\square_S) = \max_i\{\rho(\square_S, \mathcal{G}_i)\}}$. Similarly, for a profile M, we let $\rho(M) = \max_i\{\rho(M, \mathcal{G}_i)\}$.*

As before, we allow $\boldsymbol{R(N)}$ to be the number of nodes of any of $\mathcal{G}_i$ for a problem instance of size N.

The above definition is a natural generalization of the notion of an ***S*-*span*** which is defined on a ***red pebble game*** [28].

Definition 8.4 (From [28])**.** *Given a DAG of computation, G, the **red pebble game** is played using the following rules.*

(Initialization) A pebble can be placed on an input node at any time.

(Computation Step) A pebble can be placed on (or moved to) any non-input node only if all its immediate predecessors carry pebbles.

(Pebble Deletion) A pebble can be removed at any time.

(Goal) Each output node must be pebbled at least once.

The red pebble game is an abstraction of data transfer and computation in a two-level memory hierarchy. A pebble placement on an input/output node is akin to reading/output a portion of the input/output data. A pebble placement on an internal nodes corresponds to a computation step in the DAG G that is done in fast memory. Removal of a pebble models the erasure or overwriting of the value associated with the node on which the pebble resides from the fast memory.

Definition 8.5 (From [28])**.** *Given a computation DAG, G, the **S-span** of G, is the maximum number of nodes of G that can be pebbled with S pebbles in the red pebble game maximized over all initial placements of S red pebbles. (The initialization rule is disallowed.)*

Definition 8.6 (From [20])**.** *Consider a DAG of computation G. We refer to node-disjoint paths from input nodes to output nodes as* ***lines****.*

We say that the ***information speed function*** *is $\Omega(F(d))$ if for any two nodes u, v on the same line that are at least d apart, there are $F(d)$ nodes in G satisfying the following two properties.*

(a) *None of these nodes belong to the same line.*

(b) *Each of these nodes belongs to a path connecting u and v.*

Many DAM lower bound proofs use the machinery of *information speed function* to lower bound the I/O complexity of algorithms.

In the following lemma, we give a tool to transform the bounds on the *information speed function* into bounds on the *S-span*. Its proof is a restructuring of the argument of Theorem 5.1 in [20]. This transformation allows us to seamlessly *port* some DAM lower bounds to progress bounds in the cache-adaptive model. Later, we exhibit this *porting* for LCS (see Lemma 8.23) and Jacobi Multipass Filter (see Lemma 8.28) problems.

Lemma 8.7. *For any DAG, G, where all input nodes can reach all output nodes through* lines *(Definition 8.6), if the information speed function is $\Omega(F(d))$ where F is monotonically increasing and F^{-1} exists, then S-span of G is $O(SF^{-1}(S))$.*

Proof. Let I_S be any initial placement of S pebbles, and let $\text{RPG}(I_S)$ denote the nodes that could be pebbled in the *red pebble game* using the S pebbles in I_S.

First, note that the nodes of $\text{RPG}(I_S)$ must be on at most S lines, because there are initially S pebbles on nodes and lines are node-disjoint.

Claim 8.8. $\text{RPG}(I_S)$ *has at most $F^{-1}(S)+1$ nodes on any* line.

Proof. Suppose that the claim is false for some line. Then on this line there are two nodes u and v in $\text{RPG}(I_S)$ that are $F^{-1}(S)+1$ apart. WLOG assume that v is a decedent of u.

Hence, there should be $z = F(F^{-1}(S)+1)$ nodes satisfying properties (a) and (b) in Definition 8.6. Because F is monotone increasing we have $z > S$.

We argue that all of these z nodes must be in $\text{RPG}(I_S)$. Because they satisfy property (a) they are on some path from u to v. But any valid pebbling in the red pebble game should pebble *all* the nodes on *all* paths that connect u and v to be able to pebble v.

However, by property (a), these z nodes must belong to distinct lines. But this is a contradiction, because $z > S$ and nodes of $\text{RPG}(I_S)$ can be on at most S lines. □

The lemma follows from the above claim, because nodes of $\text{RPG}(I_S)$ are on at most S lines. So $|\text{RPG}(I_S)| = O(SF^{-1}(S))$. □

8.1 A Progress Bound for the Naïve Matrix Multiplication Problem

The matrix multiplication problem is formally defined as follows.

Definition 8.9. *The matrix multiplication problems is concerned with computing $C = A \times B$, where*

$$C_{ij} = \sum_k A_{ik} \times B_{kj}. \tag{7}$$

For simplicity we assume that A, B, and C are all $n \times n$ matrices with $N = n^2$ total entries.

8.1.1 A Cache-Oblivious Matrix Multiplication Algorithm

We begin with an exposition of the MM-Inplace cache-oblivious recursive matrix multiplication algorithm of [17]. Algorithm MM-Inplace computes the eight matrix subproducts "in place", adding the results of elementary multiplications into the output matrix, see Algorithm 2.

We argue that the matrix multiplication problem has a progress function ρ and progress requirement function $R(N)$ which together constitute a progress bound for the matrix multiplication problem.

Frigo et al. [17] showed that the MM-Inplace algorithm is optimal in the DAM model among algorithms that multiply two $n \times n$ matrices using just inner products to compute entries in the product matrix , i.e. each C_{ij} is computed by using Equation (7). The additions in these inner products are allowed to be performed in any order. We refer to the class of these algorithms as ***Naïve-MM***.

Several authors have established a lower bound for the Naïve-MM algorithms. Hong and Kung [20] give a lower bound on the I/O complexity of Naïve-MM algorithms by analyzing the powerful ***red-blue pebble game*** technique on the computation DAG of any Naïve-MM algorithm. Savage [28] also gives a similar lower bound using the ***red pebble game***. Finally, Irony et al. [21] give the same lower bound using a geometric argument that bounds the maximum number of elementary multiplications that can be done in a fixed memory of size Z and Z/B I/Os.

Definition 8.10. *We define $\boldsymbol{\rho_\mu}$ and $\boldsymbol{R_\mu}$ to be the ρ and R of Definition 8.2 defined on the family of DAGs of Naïve-MM algorithms.*

Savage [28] proves the following lemma which upper-bounds the size of an S-span for Naïve-MM algorithms.

Lemma 8.11 (From [28])**.** *If G is a DAG of any Naïve-MM algorithm, an S-span of G is at most $2S^{3/2}$, when $S < N$ (the input matrices do not fit in memory).*

Lemma 8.12. *We have that $\rho_\mu(\square_S) = \Theta(S^{3/2})$ and $R_\mu(N) = \Theta(N^{3/2})$. Moreover, ρ_μ and R_μ constitute a* progress bound *for the Naïve-MM problem.*

Proof. We first show that $\rho_\mu(\square_S) = \Theta(S^{3/2})$ for the class of Naïve-MM algorithms. Let $\mathcal{G}_i$ be the family of DAGs in Naïve-MM. By Lemma 8.11, we have that S-span of any of DAGs in $\mathcal{G}_i$ is at most $2S^{3/2}$. Since any computation of nodes of a DAG corresponds to a pebbling strategy in the red pebble game, we have that $\rho_\mu(\square_S) \leq 2S^{3/2}$.

As exhibited by the MM-Inplace algorithm, some DAGs in $\mathcal{G}_i$ correspond to recursive evaluations of Equation (7). Consider one of these DAGs, $\mathcal{F}$. We argue that $\rho_\mu(\square_S, \mathcal{F}) = \Omega(S^{3/2})$. Since, ρ is maximized over all DAGs in $\mathcal{G}_i$, we have that $\rho_\mu(\square_S) \geq \rho_\mu(\square_S, \mathcal{F}) = \Omega(S^{3/2})$.

Consider a recursive evaluation of Equation (7) and a level of recursion in which two submatrices A_1 and B_1 of size

Algorithm 2 Cache-oblivious matrix multiply with $O(1)$ additive overhead. In this code, A, B, and C are passed by reference.

```
function MM-INPLACE($n$,$i$,$j$,$k$,$C$,$A$,$B$)
    if $N = 1$ then
        $C[i][k] \leftarrow A[i][j] \times B[j][k]$
    else
        $i' \leftarrow i + \frac{n}{2}$; $j' \leftarrow j + \frac{n}{2}$; $k' \leftarrow k + \frac{n}{2}$
        MM-INPLACE($n/2, i, j, k, C, A, B$)
        MM-INPLACE($n/2, i, j, k', C, A, B$)
        MM-INPLACE($n/2, i', j, k, C, A, B$)
        MM-INPLACE($n/2, i', j, k', C, A, B$)
        MM-INPLACE($n/2, i, j', k, C, A, B$)
        MM-INPLACE($n/2, i, j', k', C, A, B$)
        MM-INPLACE($n/2, i', j', k, C, A, B$)
        MM-INPLACE($n/2, i', j', k', C, A, B$)
    end if
end function
```

$\left(S^{1/2}/3\right) \times \left(S^{1/2}/3\right)$ are multiplied. Both A_1 and B_1 can be brought into memory using $\leq 2S/9B$ I/Os and they fit together in a memory of size S. Multiplying them in memory takes no extra I/Os and writing the inputs and the result back takes at most $S/3B$ I/Os. To perform the multiplication $\Theta\left(\left(S^{1/2}/3\right)^3\right) = \Theta(S^{3/2})$ operations of $\mathcal{F}$ must be completed. Therefore, we have that $\rho_\mu(\square_S, \mathcal{F}) = \Omega(S^{3/2})$.

***Functions ρ_μ and R_μ constitute a* progress bound.**
We have that the size of each DAG of computation in $\mathcal{G}_i$ for a problem of size N is the same number $R_\mu(N) = \Theta(N^{3/2})$, because Equation (7) contains $R_\mu(N) = \Theta(N^{3/2})$ multiplications and additions.

Consider any DAG $\mathcal{E}$ in $\mathcal{G}_i$. If $\rho_\mu(M, \mathcal{E}) < R_\mu(N)$ for a profile M, then no algorithm whose DAG is $\mathcal{E}$ can compute all nodes of the $\mathcal{E}$ in M. Thus, if $\rho_\mu(M) = \max_i\{\rho_\mu(\square_S, \mathcal{G}_i)\}$ is less than $R_\mu(N)$, no algorithm can solve all problem instances of size N in M.

We must also exhibit *square-additivity* (Definition 3.8) and *monotonicity* (Definition 3.7) of ρ_μ. **Square-additivity** We have already established that $\rho_\mu(\square_S) = \Theta(S^{3/2})$, therefore $\rho_\mu(\square_S)$ is bounded by a polynomial in S. Consider the profile $Z = \square_{z_1} \| \dots \| \square_{z_k}$.

We first argue that $\rho_\mu(Z) \leq \sum_\ell \rho_\mu(\square_{z_\ell})$. The memory available at the beginning of $\square_{z_{\ell+1}}$ is at most $z_{\ell+1}$. By definition, no matter the initial content of cache after the execution of an algorithm on $\square_{z_1} \| \dots \| \square_{z_\ell}$, the maximum number of nodes of any of $\mathcal{G}_i$s computeable in $\square_{z_{\ell+1}}$ by using $z_{\ell+1}$ memory and $z_{\ell+1}/B$ I/Os is $\rho_\mu(\square_{z_{\ell+1}})$. Therefore, we have that $\rho_\mu(Z) \leq \sum_\ell \rho_\mu(\square_{z_\ell})$.

Next, we argue that $\rho_\mu(Z) = \Omega\left(\sum_\ell \rho_\mu(\square_{Z_\ell})\right)$. Remember that $\mathcal{F}$ is a DAG in $\mathcal{G}_i$ which corresponds to a recursive evaluation of Equation (7). We already have argued that $\rho_\mu(\square_{z_i}, \mathcal{F}) = \Omega(S^{3/2}) = \Omega(\rho_\mu(\square_{z_i}))$.

Now, we show that $\rho_\mu(Z, \mathcal{F}) = \Omega\left(\sum_\ell \rho_\mu(\square_{z_\ell}, \mathcal{F})\right) = \Omega\left(\sum_\ell \rho_\mu(\square_{z_\ell})\right)$. Because, ρ_μ is defined to be the maximum over all DAGs in $\mathcal{G}_i$, we have that $\rho_\mu(Z) \geq \rho_\mu(Z, \mathcal{F}) = \Omega\left(\sum_\ell \rho_\mu(\square_{z_\ell})\right)$.

For each square $\square_{z_\ell}$, we consider the recursive evaluation of Equation (7) at a level in which two submatrices A_ℓ and B_ℓ of size $\left(z_\ell^{1/2}/3\right) \times \left(z_\ell^{1/2}/3\right)$ are multiplied. As before, one can see that multiplying A_ℓ and B_ℓ can be done in $\square_{z_\ell}$ at it includes computing $\Omega(z_\ell^{3/2})$ nodes of $\mathcal{F}$.

Since ρ_μ is also taken as the maximum among *all* problem instances, there exists a (possibly huge) problem instance, I, such that for each $\square_{z_\ell}$, there are subproblems (A_ℓ and B_ℓ) in I that are *untouched* (not computed) in any of any of the previous squares ($\square_{z_t}$ for $t < \ell$). Therefore, we get that $\rho_\mu(Z, \mathcal{F}) = \Omega\left(\sum_\ell \rho_\mu(\square_{z_\ell}, \mathcal{F})\right) = \Omega\left(\sum_\ell \rho_\mu(\square_{z_\ell})\right)$.

Monotonicity Let $M \prec U$ be two memory profiles. Then, we have $M = L_1\|\dots\|L_k$ and $U = U_0\|U_1\|\dots\|U_k$, such that U_i is both above L_i and has a longer duration. Since for each U_i is above L_i and has a longer duration than it, it must be the case that maximum number of nodes computable in U is bigger or equal than those in M. $\square$

8.2 A Progress Bound for the All Pairs Shortest Paths Problem

We begin by defining the ***All Pairs Shortest Paths (APSP)*** problem.

Definition 8.13. *Given a weighted graph Q, that does not have negative cycles, the* ***All Pairs Shortest Paths (APSP)*** *problem is concerned with finding the length (sum of weights on edges) of shortest paths between* all *pairs of nodes.*

The dynamic programming solution of Floyd-Warshall [15, 30] to the APSP problem is based on the following recursive relation.

Allow $\mathrm{SP}(i, j, k)$ be the shortest possible path from i to j using nodes only from the set $\{1, 2, \dots, k\}$ as intermediate points along the way. Let $w(i, j)$ be the weight of the edge between nodes i and j. $\mathrm{SP}(i, j, k)$ satisfies the recursive relation

$$\begin{aligned} \mathrm{SP}(i,j,0) &= w(i,j) \\ \mathrm{SP}(i,j,k+1) &= \min\{\mathrm{SP}(i,j,k), \\ &\qquad \mathrm{SP}(i,k+1,k) + \mathrm{SP}(k+1,j,k)\}. \end{aligned} \tag{8}$$

8.2.1 A Cache-Oblivious Floyd-Warshall APSP Algorithm

Park et al. [26] give a recursive cache-oblivious algorithm for the APSP problem, FW-APSP, see Algorithm 3. They exhibit that FW-APSP is optimal in the DAM model among all algorithms that solve the APSP problem by computing the $\Theta(N^3)$ operations needed to implement the type of computation defined by Equation (8). We refer to the class of such algorithms as ***Naïve-APSP***.

Definition 8.14. *We define $\boldsymbol{\rho_\alpha}$ and $\boldsymbol{R_\alpha}$ to be the ρ and R of Definition 8.2 defined on the family of DAGs of Naïve-APSP algorithms.*

Lemma 8.15. *Family of DAGs produced by algorithms in Naïve-APSP is a subfamily of DAGs produced by algorithms in Naïve-MM.*

Proof. We claim that the data dependencies of Equation (8) is equivalent to a particular ordering on the additions and multiplications in Equation (7). To see this, map the *addition/min* nodes in DAGs of Naïve-APSP to the *multiplication/addition* nodes in DAGs of Naïve-MM respectively.

In this interpretation Equation (8) is a computation of Equation (7), where $A_{it} \times B_{tj}$ must be computed and added to the previous value for C_{ij} before any of $A_{i\ell} \times B_{\ell j}$ multiplications for all $\ell > t$.

This means that the each DAG produced by a Naïve-APSP algorithm has a corresponding DAG in Naïve-MM. □

Lemma 8.16. *If G is a DAG of any Naïve-APSP algorithm, an S-span of G is at most $O(S^{3/2})$, when $S < N^2$ (the output matrix does not fit in memory).*

Proof. The statement is a direct corollary of Lemma 8.15 and Lemma 8.11. □

Lemma 8.17. *We have that $\rho_\alpha(\Box_S) = \Theta(S^{3/2})$ and $R_\alpha(N) = \Theta(N^{3/2})$. Moreover, ρ_α and R_α constitute a* progress bound *for the Naïve-APSP problem.*

Proof. We have that the size of each DAG of computation in Naïve-APSP for a problem of size N is the same number $R_\alpha(N) = \Theta(N^{3/2})$, because Equation (8) contains $R_\alpha(N) = \Theta(N^{3/2})$ additions and min operations.

The rest of the proof is almost identical to the proof of Lemma 8.12 except that we utilize Lemma 8.16 to bound $\rho_\alpha(\Box_S)$ from above. □

8.3 Progress Bounds for the LCS and Edit Distance problems

We begin by a short exposition of the Longest Common Subsequence problem.

Definition 8.18. *A sequence $X = (x_1, \ldots, x_m)$ is a* ***subsequence*** *of sequence $Y = (y_1, \ldots, y_n)$ if there exists a strictly increasing function $f : [1, m] \to [1, n]$ such that for all $i \in [1, m]$, $x_i = y_{f(i)}$.*

Definition 8.19. *Given two sequences X and Y, the* ***Longest Common Subsequence (LCS)*** *problem is concerned with finding the longest subsequence of X that is also a subsequence of Y.*

The dynamic programming solution [13] to the LCS problem is based on the following recursive relation. Given sequences $X = (x_1, \ldots, x_m)$ and $Y = (y_1, \ldots, y_n)$, define $\boldsymbol{c[i,j]}$ to be the length of the LCS for $(x_1, \ldots, x_i)$ and $(y_1, \ldots, y_j)$. $c[i,j]$ can be computed from the following recursive relation

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{c[i-1, j], & \\ \qquad c[i, j-1]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases} \tag{9}$$

A naive implementation of the above recursion computes the table c in a row-major or column-major order and is very cache inefficient, as it incurs $\Theta(mn/B)$ I/Os.

We give a short description of the Edit Distance problem as well.

Definition 8.20. *Given two sequences X and Y, the* ***Edit Distance*** *problem is concerned with finding the smallest cost* ***edit sequence*** *that transforms X to Y.*

The ***edit*** *operations are:* $\mathbf{delete}(\boldsymbol{x_i})$ *of cost* $\boldsymbol{D(x_i)}$ *that deletes x_i from X,* $\mathbf{insert}(\boldsymbol{y_j})$ *of cost* $\boldsymbol{I(y_j)}$ *that inserts y_j into X, and* $\mathbf{substitute}(\boldsymbol{x_i, y_j})$ *of cost* $\boldsymbol{S(x_i, y_j)}$ *that replaces x_i with y_j in X.*

Chowdhury and Ramachandran in [11] describe a cache-oblivious optimal recursive algorithm for the LCS problem that also works for the Edit Distance problem by changing the character transformation cost function.

8.3.1 The Recursive LCS (and Edit Distance) Algorithm

The Recursive-LCS dynamic programming [11] is described as the composition of two recursive functions. We refer the interested reader for a complete exposition to [11]. For simplicity, we describe a high-level description of these functions when the size of $|X| = |Y| = N$.

The first function, LCS-Output-Bounary, is a recursive algorithm that on a problem of size N

(i) makes 4 recursive calls on subproblems of size $N/2$;

(ii) and besides recursive calls makes $O(1)$ additional memory references.

The second function, Recursive-LCS, is a recursive algorithm that on a problem of size N

(i) makes 3 recursive calls to Recursive-LCS on subproblems of size $N/2$;

(ii) makes 3 calls to LCS-Output-Bounary on subproblems of size $N/2$;

(iii) and makes a linear scan of size $\Theta(N)$.

Chowdhury and Ramachandran [11] prove that Recursive-LCS is optimal in DAM model among all algorithms that execute the $\Theta(N^2)$ operations needed to implement the type of computation defined by Equation (9).

All the algorithms in this class share a unique DAG of computation, G_ψ that is described by the data dependencies of Equation (9), see Figure 3.

Chowdhury and Ramachandran also prove that the Edit-Distance algorithm, which is derived by replacing the cost function of Recursive-LCS, is optimal in DAM among a similar class of algorithms. Thus, both of these problems share a unique DAG of computation, G_ψ.

Algorithm 3 Cache-oblivious Floyd-Warshall APSP with $O(1)$ additive overhead. The initial call to the recursive algorithm passes the entire input adjacency matrix of the graph as each argument. In this code, A, B, and C are passed by reference.

```
function FW-APSP($N$,$i$,$j$,$k$,$C$,$A$,$B$)
    if $N = 1$ then
        $C[i][j] \leftarrow \min\{C[i][j], A[i][k] + B[k][j]\}$
    else
        $i' \leftarrow i + \frac{N}{2}$; $j' \leftarrow j + \frac{N}{2}$; $k' \leftarrow k + \frac{N}{2}$
        FW-APSP($N/2, i, j, k, C, A, B$)
        FW-APSP($N/2, i, j, k', C, A, B$)
        FW-APSP($N/2, i', j, k, C, A, B$)
        FW-APSP($N/2, i', j, k', C, A, B$)
        FW-APSP($N/2, i, j', k, C, A, B$)
        FW-APSP($N/2, i, j', k', C, A, B$)
        FW-APSP($N/2, i', j', k, C, A, B$)
        FW-APSP($N/2, i', j', k', C, A, B$)
    end if
end function
```

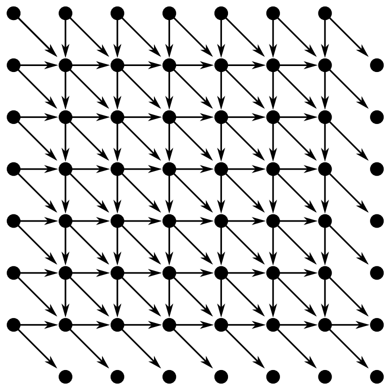

Figure 3: The DAG of computation, G_ψ, for the LCS and Edit Distance problems.

Definition 8.21. *We define* $\boldsymbol{\rho_\psi}$ *and* $\boldsymbol{R_\psi}$ *to be the* ρ *and* R *of Definition 8.2 defined on* G_ψ*.*

To prove the an upper bound for the S-span of G_ψ, we use the following bound by Chowdhury and Ramachandran [11] together with Lemma 8.7.

Lemma 8.22 (From [11])**.** *The* information speed function *for* G_ψ *satisfies* $F_{G_\psi}(d) = \Omega(d)$*. The inverse of* $F_{G_\psi}(d)$ *exists, and* $F^{-1}_{G_\psi}(d) = O(d)$*.*

Lemma 8.23. *An* S*-span of* G_ψ *is at most* $O(S^2)$*, when* $S < N$ *(the linear-size part of the dynamic programming table which the algorithm keeps track of does not fit in memory).*

Proof. The bound follows from Lemma 8.7 and the upper bound on $F^{-1}_{G_\psi}(S)$. □

Lemma 8.24. *We have that* $\rho_\psi(\square_S) = \Theta(S^2)$ *and* $R_\psi(N) = \Theta(N^2)$*. Moreover,* ρ_ψ *and* R_ψ *constitute a* progress bound *for the LCS problem.*

Proof. Computing LCS from Equation (9) for a problem of size N requires $\Theta(N^2)$ operations, so $R_\psi(N) = |G_\psi(N)| = \Theta(N^2)$.

The rest of the proof is a repetition of the argument of Lemma 8.12, except that here we only have a unique DAG G_ψ, and we utilize Lemma 8.23 to bound $\rho_\psi(\square_S)$ from above. □

8.4 A Progress Bound for the Multipass Filter problem

We begin by defining the multipass filter problem.

Definition 8.25. *A* ***one-dimensional multipass filter*** *on an array* A *of size* N *is comprised of computing values of generations* A^{t+1} *from values at generation* t *according to some update rule. A typical update function is*

$$A_i^{t+1} \leftarrow \left(A_{i-1}^t + A_i^t + A_{i+1}^t\right)/3 \qquad (10)$$

We consider computing N *generations of the update rule on* A *that has* N *elements.*

Multipass filters are used in Jacobi iteration for solving heat-diffusion equationsand simulation of lattice gases with cellular automata.

8.4.1 The Cache-Oblivious Recursive Jacobi Multipass Filter Algorithm

We give an exposition of the cache-oblivious Jacobi Multipass Filter algorithm JACOBI from [27].

The cache-oblivious JACOBI algorithm [27] is described as the composition of *three* recursive functions: JACOBI△, JACOBI▽ and JACOBI, see Algorithm 4.

Prokop [27] proves that JACOBI is optimal in DAM model among all algorithms that execute the $\Theta(N^2)$ operations needed to implement the type of computation defined by Equation (10).

All the algorithms in this class share a unique DAG of computation, G_η that is described by the data dependencies of Equation (10), see Figure 4.

Definition 8.26. *We define* $\boldsymbol{\rho_\eta}$ *and* $\boldsymbol{R_\eta}$ *to be the* ρ *and* R *of Definition 8.2 defined on* G_η*.*

To prove the an upper bound for the S-span of G_η, we use the following bound by Prokop [27] together with Lemma 8.7.

Algorithm 4 The Jacobi Multipass Filter algorithm [27].

```
function JACOBI($A$,$N$)
    JACOBI△ $(A, N, 0, N, 0)$.
    JACOBI▽ $(A, N, N/2, N, 0)$.
    JACOBI△ $(A, N, 0, N, 0)$.
    JACOBI▽ $(A, N, 0, N, N/2)$.
end function
function JACOBI▽($A$,$N$,$s$,$w$,$\tau$)
    if $w > 2$ then
        JACOBI▽ $(A, N, (s + w/4), w/2, \tau)$.
        JACOBI△ $(A, N, (s + w/4), w/2, (\tau + w/4))$.
        JACOBI▽ $(A, N, s, w/2, (\tau + w/4))$.
        JACOBI▽ $(A, N, (s + w/2), w/2, (\tau + w/4))$.
    end if
end function
```

Algorithm 5 The JACOBI△ subroutine.

```
 1: function JACOBI△($A$,$N$,$s$,$w$,$\tau$)
 2:     if $w > 2$ then
 3:         JACOBI△ $(A, N, s, w/2, \tau)$.
 4:         JACOBI△ $(A, N, (s + w/2), w/2, \tau)$.
 5:         JACOBI▽ $(A, N, (s + w/4) + 1, w/2, \tau)$.
 6:         JACOBI△ $(A, N, (s + w/4), w/2, \tau + w/4)$.
 7:     else
 8:         $p \leftarrow \tau \mod 2$.
 9:         $q \leftarrow (\tau + 1) \mod 2$.
10:         $A[p][s \mod N] \leftarrow$
                $\Big(A[q][(s-1) \mod N]+$
                    $A[q][s \mod N] + A[q][(s+2) \mod N]\Big)/3$.
11:         $A[p][(s+1) \mod N] \leftarrow$
                $\Big(A[q][s \mod N]+$
                    $A[q][(s+1) \mod N] + A[q][(s+2) \mod N]\Big)/3$.
12:     end if
13: end function
```

Lemma 8.27 (From [27])**.** *The* information speed function *for G_η satisfies $F_{G_\eta}(d) = \Omega(d)$. The inverse of $F_{G_\eta}(d)$ exists, and $F^{-1}_{G_\eta}(d) = O(d)$.*

Lemma 8.28. *An S-span of G_η is at most $O(S^2)$, when $S < N$ (the size-N temporary storage—see [27]—does not fit in memory).*

Proof. The bound follows from Lemma 8.7 and the upper bound on $F^{-1}_{G_\eta}(S)$. □

Lemma 8.29. *We have that $\rho_\eta(\square_S) = \Theta(S^2)$ and $R_\eta(N) = \Theta(N^2)$. Moreover, ρ_η and R_η constitute a* progress bound *for the multipass filter problem.*

Proof. Computing N generations of a multipass filter from 10 on an array of size N requires $\Theta(N^2)$ operations, so $R_\eta(N) = |G_\eta(N)| = \Theta(N^2)$.

The rest of the proof is a repetition of the argument of Lemma 8.12, except that here we only have a unique DAG G_η, and we utilize Lemma 8.28 to bound $\rho_\eta(\square_S)$ from above. □

8.5 A Progress Bound for the FFT Problem

We begin by defining the ***Discrete Fourier Transform (DFT)*** problem.

Definition 8.30. *Let X be an array of N complex numbers. The **Discrete Fourier Transform (DFT)** of X is an array Y defined by the formula*

$$Y[i] = \sum_{j=0}^{N-1} X[j] w_N^{-ij} \qquad i = 0, \ldots, N-1; \qquad (11)$$

where $w_N = e^{2\pi\sqrt{-1}/N}$.

DFT is the most important discrete transform, used to perform Fourier analysis in many practical applications, such as digital signal processing, image processing, solving partial differential equations, or even multiplying large integers. Directly computing Equation (11) requires $\Theta(N^2)$ operations.

A ***Fast Fourier Transform (FFT)*** algorithm is a recursive method of computing a DFT by computing only $\Theta(N \log N)$ operations.

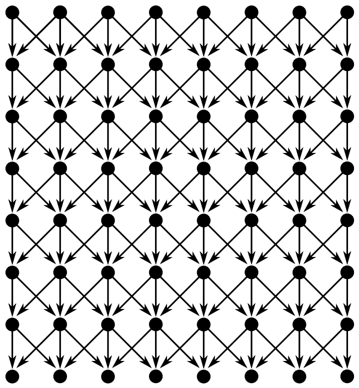

Figure 4: The DAG of computation, G_η, for the Jacobi Multipass Filter problem.

Definition 8.31. *Let $N = N_1N_2$ be any integer factorization of N. A **Cooley-Tukey Fast Fourier Transform (FFT)** algorithm [12] A computes Equation* (11) *by computing*

$$Y[i_1 + i_2N_1] = \sum_{j_2=0}^{N_2-1}\left(\left(\sum_{j_1=0}^{N_1-1} X[j_1N_2 + j_2]w_{N_1}^{-i_1j_1}\right) w_N^{-i_1j_2}\right) w_{N_2}^{-i_2j_2}; \quad (12)$$

when N is $\Omega(1)$. When $N = O(1)$, A computes Equation (11) *directly.*

*The $w_N^{-i_1j_2}$ terms are called **twiddle factors** [14].*

Note, that the inner and outer summations in Equation (12) are both DFTs. This observation helps FFT algorithms to recursively compute DFTs.

8.5.1 The Cache-Oblivious FFT Algorithm

We describe the cache-oblivious FFT algorithm, CO-FFT, due to Firgo et al. [17], see Algorithm 6.

The class of Cooley-Tukey FFT algorithms (Definition 8.31) describes a unique DAG of computation G_ϕ. Frigo et al. [17] prove that CO-FFT is optimal in DAM model among all Cooley-Tukey FFT algorithms by using a lower bound given by Hong and Kung [20], who study G_ϕ.

Definition 8.32. *We define $\boldsymbol{\rho_\phi}$ and $\boldsymbol{R_\phi}$ to be the ρ and R of Definition 8.2 defined on G_ϕ.*

Savage [28] proves the following lemma which upper-bounds the size of an S-span for G_ϕ.

Lemma 8.33 (From [28])**.** *The S-span of G_ϕ on N input nodes is at most $2S \log S$ (when $S \le n$).*

Lemma 8.34. *We have that $\rho_\phi(\square_S) = \Theta(2S \log S)$ and $R_\phi(N) = \Theta(N \log N)$ ρ_ϕ and R_ϕ constitute a* progress bound *for the Cooley-Tukey FFT problem.*

Proof. A Cooley-Tukey FFT algorithm needs to compute $R_\phi(N) = |G_\phi(N)| = \Theta(N \log N)$ operations from 12 to solve a problem of size N.

The rest of the proof is a repetition of the argument of Lemma 8.12, except that here we only have a unique DAG G_ϕ, and we utilize Lemma 8.33 to bound $\rho_\phi(\square_S)$ from above. □

9. OPTIMAL RECURSIVE CACHE-ADAPTIVE ALGORITHMS

In this chapter, we exhibit several applications of the optimality criteria theorems of Sections 6 and 7 to prove optimality of algorithms in the cache-adaptive model.

9.1 Optimal Matrix Multiplication and APSP

We can now apply Theorem 7.5 to show that MM-Inplace and FW-APSP algorithms are optimally progressing.

Theorem 9.1 strengthens the optimality result of [8] for MM-Inplace and FW-APSP as it relaxes the cache tallness assumption of $\max\{\Theta(B \log N), B^2\}$ in [8] to be just $\Theta(B^2)$.

Theorem 9.1. *The cache-oblivious matrix multiplication algorithm,* MM-Inplace *[17], and the cache-oblivious* FW-APSP *algorithm [26] are optimally progressing and cache-adaptive on $\Theta(B^2)$-tall profiles.*

Proof. Both MM-Inplace and FW-APSP require a $\Theta(B^2)$-tall cache to be optimal in the DAM model.

By Lemma 8.12, we have that $\rho_\mu(\square_X) = \Theta(X^{3/2})$ and $R_\mu(N) = \Theta(N^{3/2})$ constitute a progress bound for the naïve matrix multiplication problem. And by Lemma 8.17, we have that $\rho_\alpha(\square_X) = \Theta(X^{3/2})$ and $R_\alpha(N) = \Theta(N^{3/2})$ constitute a progress bound for the naïve APSP problem.

Since both MM-Inplace and FW-APSP are (a, b, c)-regular algorithms, and $c = 0$ we have that by Theorem 7.5, they are optimally progressing and cache-adaptive on all $\Theta(B^2)$-tall memory profiles. Note that there exists an $\varepsilon \in (0, 0.1)$ such that $\max\{\Theta(B^2), (3(1+\varepsilon)B \log_4 B)^{1+\varepsilon}\} = \Theta(B^2)$. □

9.2 Optimal LCS and Edit Distance

We use Theorem 6.12 to prove that the Recursive-LCS algorithm of Chowdhury and Ramachandran [11] (see Section 8.3.1) is optimally progressing and thus optimally cache-adaptive. Recursive-LCS is optimal in the DAM model and unlike previous algorithms, this algorithm does not require a tall cache assumption.

Theorem 9.2. *Assume that $B \ge 4$. Pick an $\varepsilon \in (0, 0.1)$ arbitrary close to 0, and let $d = 3(1+\varepsilon)$ and $\lambda = (dB \log_2 B)^{1+\varepsilon}$. For all λ-tall memory profiles, the cache-oblivious* Recursive-LCS *and* Edit-Distance *algorithms [11] are optimally progressing (and hence optimally cache-adaptive).*

Proof. Both the Recursive-LCS and LCS-Output-Boundary functions (see Section 8.3.1) have linear space complexity as they recycle the auxiliary space.

By Lemma 8.24, $\rho_\psi(\square_X) = \Theta(X^2)$ and $R_\psi(N) = \Theta(N^2)$ constitute a progress bound for the LCS problem among all algorithms that execute the $\Theta(N^2)$ operations needed to implement the type of computation defined by Eq. (9).

Since both of the subroutines of the algorithm are computing LCS (for different parts of the table), the progress bound applies to both of the functions.

We apply Theorem 6.12 by allowing $\lambda = (dB \log_2 B)^{1+\varepsilon}$. Suppose A_1, A_2 signify the LCS-Output-Boundary and Recursive-LCS respectively.

Algorithm 6 The CO-FFT algorithm [17].

```
1: function CO-FFT($R$,$N$)
2:     Pretend that the input is a row-major $\sqrt{N}\times\sqrt{N}$ matrix $R$. Perform an in-place matrix transpose operation by calling M-TRANSPOSE($R$).    ▷ At this stage, the inner sum corresponds to a DFT of the $\sqrt{N}$ rows of the transposed matrix.
3:     for each row $R_i$ of the matrix do
4:         CO-FFT($R_i$, $\sqrt{N}$).
5:     end for
6:     A linear scan to multiple each element of $R$ by a by the twiddle factors (can be computed on the fly).
7:     M-TRANSPOSE($R$), so that the inputs to the next stage are arranged in contiguous locations.
8:     for each row $R_i$ of the matrix do
9:         CO-FFT($R_i$, $\sqrt{N}$).
10:    end for                    ▷ Compute $\sqrt{N}$ DFTs of the rows of the matrix recursively.
11:    M-TRANSPOSE($R$), in place so as to produce the correct output order.
12: end function
```

We first solve $\mathcal{T}_1$, $\mathcal{U}_1$ and $\mathcal{V}_1$. Since $c_1 = 0$ and LCS-Output-Boundary does not have a subcall to Recursive-LCS, we have that:

$$\mathcal{U}_1(N) = \begin{cases} \Theta\left(\rho(\Box_N)\right) + 4\mathcal{U}_1(N/2) & \text{if } N = \Omega(\lambda),\ N^0 = \Omega(\lambda) \\ 4\mathcal{U}_1(N/2) & \text{if } N = \Omega(\lambda),\ N^0 \neq \Omega(\lambda) \\ 0 & \text{if } N \neq \Omega(\lambda); \end{cases}$$
$$= 0$$

As for $\mathcal{T}_1(N)$, we have that

$$\mathcal{T}_1(N) = \begin{cases} \max\left\{\Theta\left(N^2\right), 4\mathcal{T}(N/2)\right\} & \text{if } \lambda < N \\ \Theta\left(\lambda^2\right) & \text{if } N \leq \lambda; \end{cases}$$
$$= \Theta(N^2).$$

Because $c_1 = 0$, we have that $N^0 = O(B)$ and

$$\mathcal{V}_1(N) = \begin{cases} \Theta\left((B\log_{1/b} N)^2\right) + 4\mathcal{V}_1(N/2) \\ \qquad\qquad \text{if } B\log_{1/b} N = \Omega(\lambda) \text{ and } N^0 \leq B \\ 0 \qquad\qquad \text{if } B\log_{1/b} N \neq \Omega(\lambda). \end{cases}$$

We bound $\mathcal{V}_1(N)$ from above. Note that as long as $N_0 > \lambda \geq (dB\log_2 B)^{1+\varepsilon}$, by Lemma 7.2 we have that

$$\frac{N_0^{1/(1+\varepsilon)}}{\log_2 N_0} > B \Rightarrow N_0^{1/(1+\varepsilon)} > B\log_2 N_0.$$

And when $N_1 \leq \lambda$, we have $B\log_{1/b} N_1 \leq B\log_{1/b}\lambda < \lambda^{1/(1+\varepsilon)} < \lambda$ by another application of Lemma 7.2 because $\lambda \geq (dB\log_{1/b} B)^{1+\varepsilon}$.

Therefore, $\mathcal{V}_1(N) = O(\mathcal{Z}_1(N))$ where

$$\mathcal{Z}_1(N) = \begin{cases} \Theta\left(N^{2/(1+\delta)}\right) + 4\mathcal{Z}_1(N/2) & \text{if } N > \lambda \\ \Theta\left(\lambda^2\right) & \text{if } N \leq \lambda. \end{cases}$$
$$= O\left(N^2\right) \qquad \text{by applying the Master method.}$$

We have that $c_2 = 1$. We now solve $\mathcal{U}_2$:

$$\mathcal{U}_2 = \begin{cases} \Theta(N^2) + 3\mathcal{U}_2(N/2) + 3\mathcal{U}_1(N/2) & \text{if } N = \Omega(\lambda) \\ 0 & \text{otherwise.} \end{cases}$$
$$= \begin{cases} \Theta(N^2) + 3\mathcal{U}_2(N/2) & \text{if } N = \Omega(\lambda) \\ 0 & \text{otherwise.} \end{cases}$$
$$= \Theta(N^2).$$

As for $\mathcal{V}_2(N)$ we get

$$\mathcal{V}_2(N) = \begin{cases} 3\mathcal{V}_2(N/2) + 3\mathcal{V}_1(N/2) \\ \qquad\qquad \text{if } B\log_{1/b} N = \Omega(\lambda) \text{ and } N^1 > B \\ 3\mathcal{V}_2(N/2) + 3\mathcal{V}_1(N/2) + \Theta\left((B\log_{1/b} N)^2\right) \\ \qquad\qquad \text{if } B\log_{1/b} N = \Omega(\lambda) \text{ and } N^1 \leq B \\ 0 \qquad\qquad \text{otherwise.} \end{cases}$$

Note that when $N^1 \leq B$, $B\log_2 N \leq B\log_{1/b} B$. Because $\lambda \geq (dB\log_2 B)^{1+\varepsilon}$, we have that $B\log_2 N \neq \Omega(\lambda)$. Therefore the middle case in $\mathcal{V}_2(N)$ never happens.

$$\mathcal{V}_2(N) = \begin{cases} 3\mathcal{V}_2(N/2) + 3\mathcal{V}_1(N/2) \\ \qquad\qquad \text{if } B\log_{1/b} N = \Omega(\lambda) \text{ and } N^1 > B \\ 0 \qquad\qquad \text{if } B\log_{1/b} N \neq \Omega(\lambda). \end{cases}$$
$$= \begin{cases} 3\mathcal{V}_2(N/2) + O(N^2) \\ \qquad\qquad \text{if } B\log_{1/b} N = \Omega(\lambda) \text{ and } N^1 > B \\ 0 \qquad\qquad \text{if } B\log_{1/b} N \neq \Omega(\lambda). \end{cases}$$
$$= O(N^2) \qquad \text{by applying the Master method.}$$

Finally, we show that $\mathcal{T}_2(N) = \Theta(N^2)$.

$$\mathcal{T}_2(N) = \begin{cases} \max\left\{\Theta(N^2), 3\mathcal{T}_2(N/2) + 3\mathcal{T}_1(N/2)\right\} & \text{if } N > \lambda \\ \Theta(\lambda^2) & \text{if } N \leq \lambda. \end{cases}$$
$$= \Omega(N^2).$$

We also have that $\mathcal{T}_2(N) < X(N)$, where

$$X(N) = \begin{cases} 3X(N/2) + \Theta(N^2) + 3\mathcal{T}_1(N/2) & \text{if } N > \lambda \\ \Theta(\lambda^2) & \text{if } N \leq \lambda. \end{cases}$$
$$= \begin{cases} 3X(N/2) + \Theta(N^2) & \text{if } N > \lambda \\ \Theta(\lambda^2) & \text{if } N \leq \lambda. \end{cases}$$
$$= O(N^2).$$

Hence, we have concluded that $\mathcal{T}_2(N) + \mathcal{U}_2(N) + \mathcal{V}_2(N) = \Theta(N^2)$. Therefore, $\rho_\psi(W_{LCS,N,\lambda}) = O(N^2)$.

Since $\rho_\psi(W_{LCS,N,\lambda}) = O(N^2) = O(R_\psi(N))$, Recursive-LCS is optimally progressing and optimally cache-adaptive. Since Edit-Distance has the same progress bound, the above argument holds for it as well. □

9.3 Optimal Jacobi Multipass Filter

We use Theorem 6.12 to prove that the JACOBI algorithm (Algorithm 4) is optimally progressing and optimally cache-adaptive.

Theorem 9.3. *Assume that $B \geq 4$. Pick an $\varepsilon \in (0, 0.1)$ arbitrary close to 0, and let $d = 3(1+\varepsilon)$ and $\lambda = (dB\log_2 B)^{1+\varepsilon}$. For all λ-tall memory profiles, the cache-oblivious* JACOBI *algorithm [27] is optimally progressing (and hence optimally cache-adaptive).*

Proof. The cache-oblivious JACOBI algorithm [27] is described as the composition of *three* recursive functions: JACOBI$\triangle$, JACOBI∇ and JACOBI, see Algorithm 4.

JACOBI$\triangle$ is a recursive algorithm that on a problem of size N

(i) makes 3 calls to JACOBI$\triangle$ on subproblems of size $N/2$;

(ii) makes 1 call to JACOBI∇ on subproblems of size $N/2$;

(iii) and besides recursive calls makes $O(1)$ additional memory references.

JACOBI∇ is a recursive algorithm that on a problem of size N

(i) makes 3 calls to JACOBI∇ on subproblems of size $N/2$;

(ii) makes 1 call to JACOBI$\triangle$ on subproblems of size $N/2$;

(iii) and besides recursive calls makes $O(1)$ additional memory references.

Finally, JACOBI is a recursive algorithm that on a problem of size N

(i) makes 2 calls to JACOBI∇ on subproblems of size $N/2$;

(ii) makes 2 calls to JACOBI$\triangle$ on subproblems of size $N/2$;

(iii) and besides recursive calls makes $O(1)$ additional memory references.

The JACOBI algorithm is computing N generations of the update rule in Definition 8.25, on an array A of size N. And the algorithm in [27] is described as if, it computes the whole $N \times N$ matrix of N generations. However, JACOBI can be adapted to use only one auxiliary array of size N in addition to the original array A. Thus, the space complexity of all the subroutines in the JACOBI algorithm is linear.

Lemma 8.29 shows that $\rho_\eta(\Box_X) = \Theta(X^2)$ and $R_\eta(N) = \Theta(N^2)$ constitute a progress bound for the multipass filter problem among all algorithms that execute the $\Theta(N^2)$ operations needed to implement the type of computation defined by Eq. (10).

Since all three subroutines of the algorithm are computing the computations of Eq. (10), the progress bound applies to all three of the functions.

We apply Theorem 6.12 by allowing $\lambda = (dB\log_2 B)^{1+\varepsilon}$. Let A_1, A_2, A_3 represent JACOBI$\triangle$, JACOBI∇ and JACOBI respectively.

By simultaneously solving $\mathcal{T}_1(N)$ and $\mathcal{T}_2(N)$, we get that

$$\mathcal{T}_1(N) = \begin{cases} \max\left\{\Theta(N^2), 3\mathcal{T}_1(N/2) + \mathcal{T}_2(N/2)\right\} & \text{if } \lambda < N \\ \Theta(\lambda^2) & \text{if } N \leq \lambda. \end{cases}$$
$$= \Theta(N^2);$$

$$\mathcal{T}_2(N) = \begin{cases} \max\left\{\Theta(N^2), 3\mathcal{T}_2(N/2) + \mathcal{T}_1(N/2)\right\} & \text{if } \lambda < N \\ \Theta(\lambda^2) & \text{if } N \leq \lambda. \end{cases}$$
$$= \Theta(N^2).$$

Therefore, for $\mathcal{T}_3(N)$, we have

$$\mathcal{T}_3(N) = \begin{cases} \max\left\{\Theta(N^2), 2\mathcal{T}_1(N/2) + 2\mathcal{T}_2(N/2)\right\} & \text{if } \lambda < N \\ \Theta(\lambda^2) & \text{if } N \leq \lambda \end{cases}$$
$$= \Theta(N^2).$$

Since $c_1 = c_2 = c_3 = 0$, we have that $N^{c_i} \neq \Omega(\lambda)$ for $i = 1, 2, 3$. Therefore,

$$\mathcal{U}_1(N) = \begin{cases} 3\mathcal{U}_1(N/2) + \mathcal{U}_2(N/2) & \text{if } N = \Omega(\lambda),\ N^0 \neq \Omega(\lambda) \\ 0 & \text{otherwise.} \end{cases}$$

$$\mathcal{U}_2(N) = \begin{cases} 3\mathcal{U}_2(N/2) + \mathcal{U}_1(N/2) & \text{if } N = \Omega(\lambda),\ N^0 \neq \Omega(\lambda) \\ 0 & \text{otherwise.} \end{cases}$$

$$\mathcal{U}_3(N) = \begin{cases} 2\mathcal{U}_1(N/2) + 2\mathcal{U}_2(N/2) & \text{if } N = \Omega(\lambda),\ N^0 \neq \Omega(\lambda) \\ 0 & \text{otherwise.} \end{cases}$$

We conclude that $\mathcal{U}_1(N) = \mathcal{U}_2(N) = \mathcal{U}_3(N) = 0$.

Since $c_1 = c_2 = c_3 = 0$, we have that $N^{c_i} = O(B)$ for $i = 1, 2, 3$. Therefore,

$$\mathcal{V}_1(N) = \begin{cases} 3\mathcal{V}_1(N/2) + \mathcal{V}_2(N/2) \\ \qquad \text{if } B\log_{1/b} N = \Omega(\lambda) \text{ and } N^0 > B \\ \Theta\left((B\log_{1/b} N)^2\right) + 3\mathcal{V}_1(N/2) + \mathcal{V}_2(N/2) \\ \qquad \text{if } B\log_{1/b} N = \Omega(\lambda) \text{ and } N^0 \leq B \\ 0 \qquad \text{if } N \neq \Omega(\lambda); \end{cases}$$

$$\mathcal{V}_2(N) = \begin{cases} 3\mathcal{V}_2(N/2) + \mathcal{V}_1(N/2) \\ \qquad \text{if } B\log_{1/b} N = \Omega(\lambda) \text{ and } N^0 > B \\ \Theta\left((B\log_{1/b} N)^2\right) + 3\mathcal{V}_2(N/2) + \mathcal{V}_1(N/2) \\ \qquad \text{if } B\log_{1/b} N = \Omega(\lambda) \text{ and } N^0 \leq B \\ 0 \qquad \text{if } N \neq \Omega(\lambda); \end{cases}$$

$$\mathcal{V}_3(N) = \begin{cases} 2\mathcal{V}_1(N/2) + 2\mathcal{V}_2(N/2) \\ \qquad \text{if } B\log_{1/b} N = \Omega(\lambda) \text{ and } N^0 > B \\ \Theta\left((B\log_{1/b} N)^2\right) + 2\mathcal{V}_2(N/2) + 2\mathcal{V}_1(N/2) \\ \qquad \text{if } B\log_{1/b} N = \Omega(\lambda) \text{ and } N^0 \leq B \\ 0 \qquad \text{if } N \neq \Omega(\lambda). \end{cases}$$

Note that as long as $N_0 > \lambda \geq (dB\log_2 B)^{1+\varepsilon}$, by Lemma 7.2 we have that

$$\frac{N_0^{1/(1+\varepsilon)}}{\log_2 N_0} > B \Rightarrow N_0^{1/(1+\varepsilon)} > B\log_2 N_0.$$

And when $N_1 \leq \lambda$, we have $B\log_{1/b} N_1 \leq B\log_{1/b}\lambda < \lambda^{1/(1+\varepsilon)} < \lambda$ by another application of Lemma 7.2 because $\lambda \geq (dB\log_{1/b} B)^{1+\varepsilon}$.

Algorithm 7 Cache-oblivious matrix transpose [17]. The algorithm writes A^T into the output matrix B.

```
function M-TRANSPOSE($A$, $B$)
    if size($A$) = 1 then
        Write $A$ into $B$.
    else
        M-TRANSPOSE($A_1, B_1$).
        M-TRANSPOSE($A_2, B_3$).
        M-TRANSPOSE($A_3, B_2$).
        M-TRANSPOSE($A_4, B_4$).
    end if
end function
```

Therefore, we have that $\mathcal{V}_1(N) = O(\mathcal{Z}_1(N))$ and $\mathcal{V}_2(N) = O(\mathcal{Z}_2(N))$ where

$$\mathcal{Z}_1(N) = \begin{cases} \Theta\left(N^{2/(1+\delta)}\right) + 3\mathcal{Z}_1(N/2) + \mathcal{Z}_2(N/2) & \text{if } N > \lambda \\ \Theta(\lambda^2) & \text{if } N \le \lambda \end{cases}$$

$$\mathcal{Z}_2(N) = \begin{cases} \Theta\left(N^{2/(1+\delta)}\right) + 3\mathcal{Z}_2(N/2) + \mathcal{Z}_1(N/2) & \text{if } N > \lambda \\ \Theta(\lambda^2) & \text{if } N \le \lambda \end{cases}$$

By manually solving the recursions for $Z_1(N)$ and $Z_2(N)$ simultaneously, we get that $\mathcal{Z}_1(N) = O(N^2)$ and $\mathcal{Z}_2(N) = O(N^2)$.

Finally, we have that $\mathcal{V}_3(N) = O(\mathcal{Z}_3(N))$, where

$$\mathcal{Z}_3(N) = \begin{cases} \Theta\left(N^{2/(1+\delta)}\right) + 2\mathcal{V}_1(N/2) + 2\mathcal{V}_2(N/2) & \text{if } N > \lambda \\ \Theta(\lambda^2) & \text{if } N \le \lambda. \end{cases}$$
$$= O\left(N^2\right).$$

Therefore, $\rho_\eta(W_{Jacobi,N,\lambda}) = O(\mathcal{T}_3(N) + \mathcal{U}_3(N) + \mathcal{V}_3(N)) = \Theta(N^2)$. Since $\rho_\eta(W_{Jacobi,N,\lambda}) = O(N^2) = O(R_\eta(N))$, JACOBI is optimally progressing and optimally cache-adaptive. $\square$

9.4 Optimal Matrix Transpose

Definition 9.4. *Given a matrix A, the* ***matrix transpose*** *problem is concerned with computing the matrix A^T, where $[A^T]_{ij} = [A]_{ji}$.*

We describe a cache-oblivious algorithm, M-TRANSPOSE, for the matrix transpose problem. For a matrix A, allow A_1, A_2, A_3, A_4 to be the four subquadrants of A. We have that the transpose of A, A^T, satisfies

$$A = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} \Rightarrow A^T = \begin{pmatrix} A_1^T & A_3^T \\ A_1^T & A_4^T \end{pmatrix}. \quad (13)$$

The M-TRANSPOSE algorithm is a variant of the cache-oblivious algorithm of Frigo et al. [17],see Algorithm 7. Like the algorithm in [17], M-TRANSPOSE requires a tall cache of size $\Theta(B^2)$ to be optimal in the DAM model and incurs $\Theta(N/B)$ I/Os (irrespective of the profile).

Theorem 9.5 strengthens the optimality result of [8] for M-TRANSPOSE as it relaxes the cache tallness assumption of $\max\{\Theta(B \log N), B^2\}$ in [8] to be just $\Theta(B^2)$.

Theorem 9.5. *For all $\Theta(B^2)$-tall memory profiles, the cache-oblivious matrix transpose algorithm,* M-TRANSPOSE *[17], is optimally progressing and cache-adaptive and it takes $\Theta(N/B)$ I/Os to finish.*

Proof. Let $\rho_\tau(\square_S) = S/B$ be the duration of a square $\square_S$ and $R_\tau(N) = N$ be the size of the input. ρ_τ is trivially monotone and square-additive. Furthermore, all algorithms that solve all inputs of size N, must read the input. Hence, ρ_τ and R_τ constitute a progress bound for the matrix transpose problem.

The M-TRANSPOSE is an (a, b, c)-regular algorithm with $c = 0$ and $a = 4 = 1/b$. Therefore, by Theorem 7.5 it is optimally progressing and cache-adaptive. Note that there exists an $\varepsilon \in (0, 0.1)$ such that $\max\{\Theta(B^2), (3(1+\varepsilon)B \log_4 B)^{1+\varepsilon}\} = \Theta(B^2)$.

The I/O complexity of M-TRANSPOSE is evident from the analysis of Frigo et al. [17]. $\square$

10. CACHE-OBLIVIOUS FAST FOURIER TRANSFORM

The cache-oblivious FFT algorithm of Frigo et al. [17], CO-FFT(see Algorithm 6), is not a GCR algorithm because its recursive calls are on subproblems of size $\sqrt{N}$, rather than on subproblems a constant factor smaller. Consequently, a square S may contain subproblems of size only $\Theta(\sqrt{|S|})$ instead of $\Theta(|S|)$, so the charging scheme of Theorem 6.12 does not work "out of the box".

However, a square S must either intersect a linear scan of size $\Omega(|S|)$ or contain enough subproblems of size $\Omega(\sqrt{|S|})$ so that their total size sums to $\Theta(|S|)$.

Lemma 8.34 shows that $\rho_\phi(\square_X) = \Theta(X \log X)$ and $R_\phi(N) = \Theta(N \log N)$ constitute a progress bound for Cooley-Tuckey FFT algorithms. Thus, we can divide the progress of S among all the subproblems contained in S. CO-FFT uses the cache-oblivious matrix-transpose algorithm as a subroutine, so it has a tall-cache requirement, i.e. $H(B) = \Theta(B^2)$.

Theorem 10.1. CO-FFT *is a $\Theta(\log\log(N/B^2))$ factor away from being optimally progressing on $\Theta(B^2)$-tall profiles, and a $O(\log\log(N/B^2))$ factor away from being optimally cache-adaptive.*

Lemma 10.2. *If M is an N-fitting $\Theta(B^2)$-tall profile for* CO-FFT*, then every square, S, of M satisfies at least one of the following two properties*

- *S overlaps a linear scan of size $\Omega(|S|)$.*
- *S encompasses a sequence of k executions of A on subproblems of size ℓ, where $k\ell = \Theta(|S|)$.*

Proof. The proof is the same as that of Lemma 6.9, except we unroll the recursion $i + 1$ times so that $N^{1/2^{i+1}} \le |S|/2$.

Note that because the size of linear scans in CO-FFT is linear in each invocation, and M is $\Theta(B^2)$-tall, squares of M cannot be overhead-containing squares. $\square$

Proof of Theorem 10.1. Consider a "canonical bad profile" for CO-FFT which has large boxes whenever CO-FFT performs a linear scan. The recursion for this profile is then

$$M_{FFT,N} = \begin{cases} \square_N \,\|M_{FFT,\sqrt{N}}^{\sqrt{N}}\|\, \square_N \,\|M_{FFT,\sqrt{N}}^{\sqrt{N}}\|\, \square_N & \text{if } N \ge \Theta(B^2) \\ \Theta(\square_{B^2}) & \text{otherwise.} \end{cases}$$

We have that $\rho_\phi(\square_N) = \Theta(N \log N)$, Therefore $\rho_\phi(M_{FFT,N})$ satisfies the recurrence

$$\rho_\phi(M_{FFT,N}) = \begin{cases} 2\sqrt{N}\rho_\phi(M_{FFT,\sqrt{N}}) + \Theta(N \log N) \\ \qquad\qquad\qquad\qquad \text{if } N \geq \Theta(B^2) \\ \Theta(B^2 \log B) \qquad\qquad \text{otherwise,} \end{cases}$$

which has solution $\rho_\phi(M_{FFT,N}) = \Theta(N \log N \log\log(N/B^2))$.

Thus, CO-FFT can be at least $\log\log(N/B^2)$ away from being optimally progressing.

To prove that this bound is tight, let $W_{FFT,N}$ be CO-FFT's worst case profile for inputs of size N. We will show that $\rho_\phi(W_{FFT,N}) = O(N \log N \log\log(N/B^2))$. Lemma 10.2 implies that every box S of $W_{FFT,N}$ either intersects a linear scan of size $\Omega(|S|)$ or encompasses k consecutive executions of the FFT algorithm on subproblems of size ℓ, where $kl = \Theta(|S|)$. We will account for the progress of the two cases separately.

Let $P_1(N)$ be the total progress of all the linear-scan-intersecting boxes. From the structure of the CO-FFT, P_1 satisfies the recurrence

$$P_1(N) = \begin{cases} \sqrt{N}P_1(\sqrt{N}) + \Theta(N \log N) & \text{if } N = \Omega(B^2) \\ 0 & \text{otherwise.} \end{cases}$$

This has solution $P_1(N) = O(N \log N \log\log(N/B^2))$.

Let $P_2(N)$ be the total progress of all the subproblem-encompassing boxes. Consider a box S encompassing A's execution on k successive subproblems of size ℓ. The total possible progress on S is bounded by $\Theta(k\ell \log k\ell)$. However, from the structure of the CO-FFT algorithm, we must have that $k \leq l$, so $k\ell \log k\ell = \Theta(k\ell \log \ell)$, which is the total progress from a single box placed over each of the subproblems covered by S. Since a box that covers a subproblem must cover all subproblems of that subproblem, we can bound $P_2(N)$ by finding a set of non-overlapping subproblems that cover all subproblems of the input. Hence, $P_2(N)$ can be bounded by

$$\begin{aligned} P_2(N) &= \max_{0 \leq i \leq \log\log \frac{N}{H(B)}} O(2^i N^{1-2^{-i}} N^{2^{-i}} \log N^{2^{-i}}) \\ &= O(N \log N) \end{aligned} \tag{14}$$

Therefore, whenever CO-FFT makes $R_\phi = \Theta(N \log N)$ progress on a profile, the profile can support at most $P_1(N) + P_2(N) = O(N \log N \log\log(N/B^2))$ progress. Hence, CO-FFT is $\Theta(\log\log(N/B^2))$ away from being optimally progressing and $O(\log\log(N/B^2))$ away from being optimally cache-adaptive. □

11. CONCLUSION

This paper revisits one of the most important, but also most difficult, problems in the design, analysis, and deployment of external-memory algorithms. We show that designing and analyzing cache-adaptive algorithms is tractable.

But our results have broader implications for the cache adaptive model itself.

In order for a computational model to be truly useful, (1) it needs to capture an important phenomenon that people care about, (2) its predictions have to be true-to-life, and (3) it has to be simple enough to work with. There are dozens of theoretical models capturing different aspects of memory-hierarchy performance. The DAM [1] and cache-oblivious [17, 18, 27] models have been so successful because they satisfy these criteria so effectively.

The cache-adaptive model is already known to satisfy two of these criteria. It captures the important phenomenon of changes in cache size, which can strongly affect the performance of algorithms on concurrent systems. It is true to life because it imposes no unrealistic restrictions on changes in the size of memory.

We believe that, by making it easy to analyze many algorithms in the cache-adaptive model, the tools developed in this paper provide the final piece necessary for the cache-adaptive model to join the ranks of truly useful models, such as the DAM and cache-oblivious models.

12. REFERENCES

[1] A. Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, Sept. 1988.

[2] M. Akra and L. Bazzi. On the solution of linear recurrence equations. *Computational Optimization and Applications*, 10(2):195–210, 1998.

[3] M. Akra and L. Bazzi. On the solution of linear recurrence equations. *Computational Optimization and Applications*, 10(2):195–210, 1998.

[4] R. Barve and J. S. Vitter. External memory algorithms with dynamically changing memory allocations. Technical report, Duke University, 1998.

[5] R. D. Barve and J. S. Vitter. A theoretical framework for memory-adaptive algorithms. In *Proc. 40th Annual Symposium on the Foundations of Computer Science (FOCS)*, pages 273–284, 1999.

[6] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Journal of Research and Development*, 5(2):78–101, June 1966.

[7] L. A. Belady, R. A. Nelson, and G. S. Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. *Communications of the ACM*, 12(6):349–353, 1969.

[8] M. A. Bender, R. Ebrahimi, J. T. Fineman, G. Ghasemiesfeh, R. Johnson, and S. McCauley. Cache-adaptive algorithms. In *Proc. 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 958–971, 2014.

[9] G. S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. In *Proc. 29th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 426–438. Springer-Verlag, 2002.

[10] K. P. Brown, M. J. Carey, and M. Livny. Managing memory to meet multiclass workload response time goals. In *Proc. 19th International Conference on Very Large Data Bases (VLDB)*, pages 328–328. Institute of Electrical & Electronics Engineers (IEEE), 1993.

[11] R. A. Chowdhury and V. Ramachandran. Cache-oblivious dynamic programming. In *Proc. 17th annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 591–600. ACM, 2006.

[12] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[13] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction To Algorithms*. MIT Press, 2001.

[14] P. Duhamel and M. Vetterli. Fast fourier transforms: a tutorial review and a state of the art. *Signal Processing*, 19(4):259–299, 1990.

[15] R. W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, 1962.

[16] P. Fornai and A. Iványi. FIFO anomaly is unbounded. *CoRR*, abs/1003.1336, 2010.

[17] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Annual Symposium on the Foundations of Computer Science (FOCS)*, pages 285–298, 1999.

[18] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1):4, 2012.

[19] G. Graefe. A new memory-adaptive external merge sort. Private communication, July 2013.

[20] J.-W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proc. 13th Annual ACM Symposium on the Theory of Computation (STOC)*, pages 326–333, 1981.

[21] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017–1026, 2004.

[22] R. T. Mills. *Dynamic adaptation to CPU and memory load in scientific applications*. PhD thesis, The College of William and Mary, 2004.

[23] R. T. Mills, A. Stathopoulos, and D. S. Nikolopoulos. Adapting to memory pressure from within scientific applications on multiprogrammed cows. In *Proc. 8th International Parallel and Distributed Processing Symposium (IPDPS)*, page 71, 2004.

[24] H. Pang, M. J. Carey, and M. Livny. Memory-adaptive external sorting. In *Proc. 19th International Conference on Very Large Data Bases (VLDB)*, pages 618–629. Morgan Kaufmann, 1993.

[25] H. Pang, M. J. Carey, and M. Livny. Partially preemptible hash joins. In *Proc. 5th ACM SIGMOD International Conference on Management of Data (COMAD)*, page 59, 1993.

[26] J.-S. Park, M. Penner, and V. K. Prasanna. Optimizing graph algorithms for improved cache performance. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):769–782, 2004.

[27] H. Prokop. Cache oblivious algorithms. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999.

[28] J. E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley Longman Publishing Co., Inc., 1st edition, 1997.

[29] J. S. Vitter. Algorithms and data structures for external memory. *Foundations and Trends in Theoretical Computer Science*, 2(4):305–474, 2006.

[30] S. Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, 1962.

[31] H. Zeller and J. Gray. An adaptive hash join algorithm for multiuser environments. In *Proc. 16th International Conference on Very Large Data Bases (VLDB)*, pages 186–197, 1990.

[32] W. Zhang and P.-A. Larson. A memory-adaptive sort (MASORT) for database systems. In *Proc. 6th International Conference of the Centre for Advanced Studies on Collaborative research (CASCON)*, pages 41–. IBM Press, 1996.

[33] W. Zhang and P.-A. Larson. Dynamic memory adjustment for external mergesort. In *Proc. 23rd International Conference on Very Large Data Bases (VLDB)*, pages 376–385. Morgan Kaufmann Publishers Inc., 1997.